

# **A Methodology for Detecting and Classifying Rootkit Exploits**

A Thesis

Presented to

The Academic Faculty

by

John G. Levine

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in Electrical and Computer Engineering

Georgia Institute of Technology

February 2004

Copyright ©2004 John G. Levine

# A Methodology for Detecting and Classifying Rootkit Exploits

Approved by:

Henry L. Owen, Advisor

John A. Copeland

Randal T. Abler

March 17, 2004

*To everyone who passed through  
the Antigravity and Teleportation Lab,  
you all had a part in this!*

## **Acknowledgements**

I would like to express my sincere and deep gratitude to my advisor, Dr. Henry Owen. His unwavering support and advice throughout my three years of Ph.D. study enabled me to focus on what I needed to learn and complete my studies on time.

Special thanks to Dr. Douglas Williams, Dr. John Copeland, and Dr. Randal Abler for being on my proposal and dissertation committees, and to Dr. Wenke Lee for attending my final defense.

I would like to express my gratitude to the United States Military Academy for providing me with the opportunity to attend Georgia Tech for the last three years. Never in my wildest dreams did I ever think that I would be going back to be a professor at West Point after graduating from there.

I would like to acknowledge the helpful support of Brian Culver and other personnel of the Information Security Directorate of the Georgia Tech Office of Information Technology. Your technical support and insightful discussion greatly assisted my research. I would also like to thank Lance Spiztner and Richard LaBella of the HoneyNet Alliance and Didier Contis of ECE for helping to establish the Georgia Tech HoneyNet.

I would like to thank my lab partners, past and present. A special thanks to Julian Grizzard who provide much advice and assistance as I struggled along.

Finally, I would like to thank my wife, Jackie, for never having any doubts in me.

# TABLE OF CONTENTS

Acknowledgements.....	iv
TABLE OF CONTENTS.....	v
LIST OF FIGURES .....	ix
SUMMARY .....	xii
Chapter 1 .....	1
1.1 Motivation.....	1
1.2 Dissertation Outline .....	2
Chapter 2.....	4
Origin and History of the Problem.....	4
2.1 Rootkit Description.....	5
2.2 Application Rootkits .....	8
2.3 Trojan Utility Rootkits.....	10
2.4 Kernel Rootkits .....	12
2.4.1 Kernel Level Rootkits that modify the system call table.....	15
2.4.2 Kernel Level Rootkits that redirect the system call table .....	16
2.5 Summary .....	17
Chapter 3 .....	18
Current Rootkit Detection Methodology .....	18
3.1 Checking for Rootkits using existing GPL tools .....	18
3.2 GPL Programs Intended to Maintain System Integrity.....	25
3.3 Checking for Rootkits Using Black Box Analysis .....	27

3.4 Georgia Tech Methodology for Detecting Rootkits .....	35
3.5 Summary .....	39
Chapter 4.....	40
Formal Methodology to Characterize Rootkits.....	40
4.1 Mathematical Framework to define Rootkits .....	42
4.2 Methodology to Characterize Rootkit Exploits .....	46
4.2.1 Clean Installation of Target Operating System.....	51
4.2.2 Use of a Kernel Level Debugger .....	53
4.2.3 Install File Integrity checker Program .....	54
4.2.4 Install Known Rootkit Detector Program .....	54
4.2.5 Access to /dev/kmem .....	54
4.2.6 Copy kernel text segment via /dev/kmem.....	55
4.2.7 Establish Baseline on system prior to infection with rootkit .....	56
4.2.8 Install rootkit on target system.....	58
4.2.9 Run file integrity checker.....	58
4.2.10 Run Known Rootkit Detection Program.....	60
4.2.11 Previously known rootkit identification.....	62
4.2.12 Verify Integrity of the Kernel .....	64
4.2.13 Analyze results of Kernel Integrity Check.....	68
4.2.14 Compare current kernel text code against baseline.....	70
4.3 Summary .....	72
Chapter 5 .....	73
New Methods to Detect and Classify Rootkit Exploits .....	73

5.1 Detection of Unique String Signatures in Binary System Utility Exploits.....	74
5.2 Modifications to the chkrootkit program .....	77
5.3 Detection of Unique String Signatures in Binary System Utility Exploits using String Hiding Techniques .....	78
5.4 Detection of Kernel Rootkit Exploits by Examination of System Call Table Entry Point in the Kernel .....	84
5.5 Summary .....	86
Chapter 6.....	87
Establishment of a Honeynet to Detect New Rootkit Exploits.....	87
6.1 Definition of a Honeynet .....	87
6.2 Honeynet Establishment .....	88
6.3 Data analysis .....	92
6.4 Statistical Analysis of Honeynet Traffic.....	95
6.5 Summary .....	102
Chapter 7.....	103
Detection and Analysis of a Previously Unseen Rootkit.....	103
7.1 Target System Description.....	103
7.2 Method of Compromise .....	104
7.3 Analysis Process .....	107
7.4 Rootkit Characteristics.....	115
7.5 Summary .....	117
Chapter 8.....	118
Conclusions and Further Recommendations .....	118

8.2 Contributions.....	119
8.3 Future Research .....	122
APPENDIX.....	123
A.1 The Linux Rootkit IV (lrk4).....	123
Analysis of the lrk4 login source code.....	125
A.2 The Linux Rootkit V (lrk5).....	126
A.3 The t0rn rootkit .....	128
A.4 The ark rootkit.....	131
A.5 The knark rootkit.....	133
A.6 The adore rootkit.....	138
A.7 The knark rootkit targeting the Linux 2.4 kernel .....	142
A.8 The SuckIT rootkit.....	144
A.9 The zk rootkit.....	153
References.....	158
Vita.....	164



## LIST OF FIGURES

<b>Figure 1: Previously Identified Rootkits [3] .....</b>	<b>4</b>
<b>Figure 2: filter utility for the lrk4 password sniffer program .....</b>	<b>7</b>
<b>Figure 3: level of trust in target system infected with application rootkit .....</b>	<b>9</b>
<b>Figure 4: level of trust on target system with system utility rootkit .....</b>	<b>11</b>
<b>Figure 5: level of trust on target system with kernel rootkit .....</b>	<b>14</b>
<b>Figure 6: kernel rootkit that modifies system call table.....</b>	<b>16</b>
<b>Figure 7: AIDE Configuration.....</b>	<b>19</b>
<b>Figure 8: AIDE result on kernel rootkit infected system .....</b>	<b>20</b>
<b>Figure 9: chkrootkit results on target system infected with a kernel rootkit.....</b>	<b>21</b>
<b>Figure 10: kern_check output of system infected with KNARK kernel rootkit .....</b>	<b>23</b>
<b>Figure 11: Normal system call trace of ls command.....</b>	<b>29</b>
<b>Figure 12: system call trace of lrkIV ls command .....</b>	<b>30</b>
<b>Figure 13: strace listing of lrkIV ls command.....</b>	<b>31</b>
<b>Figure 14: strace listing of clean ls command .....</b>	<b>32</b>
<b>Figure 15: system call trace of ls command on clean system .....</b>	<b>33</b>
<b>Figure 16: Flowchart of Characterization Methodology part 1 .....</b>	<b>49</b>
<b>Figure 17: Flowchart of Characterization Methodology part 2 .....</b>	<b>50</b>
<b>Figure 18: Clean Installation of Target Operating System.....</b>	<b>52</b>
<b>Figure 19: first 512 page frames of first two megabytes of kernel memory [40] .....</b>	<b>55</b>
<b>Figure 20: Baseline Establishment .....</b>	<b>57</b>
<b>Figure 21: File Integrity Check Procedures .....</b>	<b>59</b>

<b>Figure 22: Known rootkit detection analysis .....</b>	<b>61</b>
<b>Figure 23: Previously known rootkit identification .....</b>	<b>63</b>
<b>Figure 24: Verify kernel .....</b>	<b>67</b>
<b>Figure 25: Kernel Integrity Check Analysis.....</b>	<b>69</b>
<b>Figure 26: Comparison of current kernel text code .....</b>	<b>71</b>
<b>Figure 27: commands to compare login files .....</b>	<b>75</b>
<b>Figure 28: Output of fgrep function.....</b>	<b>76</b>
<b>Figure 29: rewt array used for sting hiding.....</b>	<b>81</b>
<b>Figure 30: string 'rewt' written into character array .....</b>	<b>82</b>
<b>Figure 31: The Georgia Tech Honeynet.....</b>	<b>90</b>
<b>Figure 32: ACID Output of Honeynet Alerts .....</b>	<b>93</b>
<b>Figure 33: Ethereal screen shot of Honeynet Data .....</b>	<b>94</b>
<b>Figure 34: SQL Slammer Worm .....</b>	<b>95</b>
<b>Figure 35: Microsoft RPC (port 135) exploit .....</b>	<b>96</b>
<b>Figure 36: Honeynet RPC Exploit.....</b>	<b>97</b>
<b>Figure 37: Hidden directories created by hacker .....</b>	<b>100</b>
<b>Figure 38: Modified Registry Entries on Compromised MS Machine .....</b>	<b>101</b>
<b>Figure 39: Start of Exploit .....</b>	<b>104</b>
<b>Figure 40: System Compromise Indication .....</b>	<b>105</b>
<b>Figure 41: Installation of 'r.tgz' rookit .....</b>	<b>106</b>
<b>Figure 42: AIDE results on r.tgz infected system .....</b>	<b>108</b>
<b>Figure 43: Results of kern_check program .....</b>	<b>109</b>
<b>Figure 44: strings output of r.tgz all program .....</b>	<b>111</b>

<b>Figure 45: Uninstall of INKIT kernel rootkit.....</b>	<b>112</b>
<b>Figure 46: New AIDE results on target system. ....</b>	<b>113</b>
<b>Figure 47: Accurate AIDE count of changed files .....</b>	<b>115</b>
<b>Figure 48: AIDE results on lrk4 infected system .....</b>	<b>124</b>
<b>Figure 49: AIDE results on lrk5 infected system .....</b>	<b>127</b>
<b>Figure 50: AIDE results on t0rn infected system.....</b>	<b>130</b>
<b>Figure 51: ark rootkit installation script .....</b>	<b>131</b>
<b>Figure 52: AIDE results from an ark infected system.....</b>	<b>132</b>
<b>Figure 53: chkrootkit code to detect knark lkm .....</b>	<b>135</b>
<b>Figure 54: chkrootkit output of knark infected system.....</b>	<b>135</b>
<b>Figure 55: kern_check results on knark infected system .....</b>	<b>136</b>
<b>Figure 56: <i>bvi</i> analysis of getdents system call .....</b>	<b>138</b>
<b>Figure 57: kern_check results on adore infected system.....</b>	<b>139</b>
<b>Figure 58: gdb output of adore system call .....</b>	<b>141</b>
<b>Figure 59: kern_check results on system infected with knark for Linux 2.4 .....</b>	<b>143</b>
<b>Figure 60: kdb output of system infected with knark 2.4.3 .....</b>	<b>144</b>
<b>Figure 61: chkrootkit detecting SuckIT.....</b>	<b>146</b>
<b>Figure 62: kern_check results on system infected with SuckIT .....</b>	<b>147</b>
<b>Figure 63: chkrootkit results on zk infected system .....</b>	<b>154</b>
<b>Figure 64: kern_check results on system infected with the zk rootkit.....</b>	<b>155</b>
<b>Figure 65: Unsuccessful uninstall of zk rootkit.....</b>	<b>156</b>
<b>Figure 66: Uninstall password for zk rootkit .....</b>	<b>157</b>

## SUMMARY

We propose a methodology to detect and classify rootkit exploits. The goal of this research is to provide system administrators, researchers, and security personnel with the information necessary in order to take the best possible recovery actions concerning systems that are compromised by rootkits. There is no such methodology available at present to perform this function. This may also help to detect and fingerprint additional instances and prevent further security instances involving rootkits. A formal framework was developed in order to define rootkit exploits as an existing rootkit, a modification to an existing, or an entirely new rootkit. A methodology was then described in order to apply this framework against rootkits that are to be investigated. We then proposed some new methods to detect and characterize specific types of rootkit exploits. These methods consisted of identifying unique string signatures of binary executable files as well as examining the system call table within the system kernel. We established a Honeynet in order to aid in our research efforts and then applied our methodology to a previously unseen rootkit that was targeted against the Honeynet. By using our methodology we were able to uniquely characterize this rootkit and identify some unique signatures that could be used in the detection of this specific rootkit. We applied our methodology against nine additional rootkit exploits and were able to identify unique characteristics for each of these rootkits. These characteristics could also be used in the prevention and detection of these rootkits. We determined that our methodology would fulfill a need for a method to detect and classify rootkits that are targeted against computer systems.

# Chapter 1

## 1.1 Motivation

Computers on today's Internet are vulnerable to a variety of exploits that can compromise their intended operations. Systems can be subject to denial of service attacks that prevent other computers from connecting to them for their provided service (e.g. web server) or prevent them from connecting to other computers on the Internet. They can be subject to attacks that cause them to cease operations either temporarily or permanently. A hacker may be able to compromise a system and gain root access, i.e., the ability to control that system as if the hacker were the system administrator. A hacker who gains root access on a computer system may want to maintain that access for the foreseeable future. One way for the hacker to do this is by using a rootkit. A rootkit enables the hacker to access the compromised computer system at a later time with root-level privileges. System administrators have a continuing need for techniques to determine if a hacker has installed a rootkit on their systems.

Techniques and methods currently exist to detect if a certain type of rootkit has exploited a computer systems. However, these current techniques and methods can only indicate that a system has been exploited by a rootkit. There is no indication if the rootkit exploit that has been used is a previously known rootkit or if the exploit is a modified or new rootkit. The purpose of this research is to develop a methodology to detect new rootkit exploits as well as new modifications to previously known rootkit exploits. The research includes the examination of the various types of rootkit exploits that are being developed by hackers as well as the use of a Honeynet to detect and characterize rootkit

exploits. The goal of this research is to develop a methodology that can be provided to computer security personnel to identify and characterize rootkits

## **1.2 Dissertation Outline**

The goal of the research is to develop and evaluate a methodology for detecting and classifying rootkit exploits.

Chapter 2 summarizes the fundamental concepts involved in our research including a description of the various types of rootkits that currently exist. We explore the particular characteristics of each type of rootkit. Chapter 3 is an overview of the current detection and prevention methodologies concerning rootkits. We examine general public license (GPL) tools that detect rootkits as well as programs designed to maintain system integrity. We also examine Black Box Analysis as a potential method to detect rootkits and describe the methodology used at Georgia Tech for detecting rootkits. Chapter 4 is a detailed discussion of our new formal methodology that we present as the core of our approach to detect and classify rootkit exploits. We include a mathematical framework to define rootkit exploits as well as detailed flowcharts of our methodology. Chapter 5 presents the new methods we are proposing to detect rootkit exploits. We examine methods to detect rootkits at the binary program level with the use of string signatures. We also examine a method to detect kernel level rootkits by examination of the system call entry point within the kernel. Chapter 6 is a description of the Honeynet that we have established at Georgia Tech to detect new rootkit exploits. In addition to serving as a research tool, the Georgia Tech Honeynet also helps to secure the campus network. We provide analysis on the data that has been collected by this network. Chapter 7 demonstrates the application of our methodology against a previously unseen rootkit that

was collected from the Georgia Tech HoneyNet. We conduct our analysis process against this rootkit and are able to identify specific characteristics for subsequent detections of this rootkit. Finally, Chapter 8 summarizes the contributions of this dissertation and lists possible future research topics.

## Chapter 2

### Origin and History of the Problem

Rootkits are a fairly recent phenomenon. Systems used to have utilities that could be trusted to provide a system administrator with accurate information. However, modern hackers have developed methods to conceal their activities and programs to assist in this concealment [1]. Application rootkits install a program on the target system that allow for backdoor connectivity. Trojan utility rootkits alter or replace existing system binary components. These replaced or modified programs allow backdoor access to a system as well as the ability to hide the hacker's presence on the system [2]. Kernel rootkits modify the underlying operating system kernel. Rootkits are a serious threat to the security of a computer network. Figure 1 is a listing of some currently known rootkits.

- |                              |                             |                       |
|------------------------------|-----------------------------|-----------------------|
| 01. linux root kit (lrk) 3-6 | 02. Solaris rootkit;        | 03. FreeBSD rootkit;  |
| 04. t0rn (and variants);     | 05. Ambient's Rootkit (ARK) | 06. Ramen Worm;       |
| 07. rh[67]-shaper;           | 08. RSHA;                   | 09. Romanian rootkit; |
| 10. RK17;                    | 11. Lion Worm;              | 12. Adore Worm;       |
| 13. LPD Worm;                | 14. kenny-rk;               | 15. Adore LKM;        |
| 16. ShitC Worm;              | 17. Omega Worm;             | 18. Wormkit Worm;     |
| 19. Maniac-RK;               | 20. dsc-rootkit;            | 21. Ducoci rootkit;   |
| 22. x.c Worm;                | 23. RST.b trojan;           | 24. duarawkz;         |
| 25. knark LKM;               | 26. Monkit;                 | 27. Hidrootkit;       |
| 28. Bobkit;                  | 29. Pizdakit;               | 30. t0rn v8.0;        |
| 31. Showtee;                 | 32. Optickit;               | 33. T.R.K;            |
| 34. MithRa's Rootkit;        | 35. George;                 | 36. SuckIT;           |
| 37. Scalper;                 | 38. Slapper A, B, C and D;  | 39. OpenBSD rk v1;    |
| 40. Illogic rootkit;         | 41. SK rootkit.             | 42. sebek LKM;        |
| 43. Romanian rootkit;        | 44. LOC rootkit;            | 45. shv4 rootkit;     |
| 46. Aquatica rootkit;        | 47. ZK rootkit;             | 48. 55808.A Worm;     |
| 49. TC2 Worm;                | 50. Volc rootkit;           | 51. Gold2 rootkit;    |

**Figure 1: Previously Identified Rootkits [3]**



## 2.1 Rootkit Description

A rootkit can be considered a “Trojan Horse” introduced into a computer operating system. According to [4], there are four categories of trojans: *direct masquerades*, *simple masquerades*, *slip masquerades*, and *environmental masquerades*. Direct masquerades are programs pretending to be normal programs. This type of Trojan applies to the Trojan Utility rootkit category. *Simple masquerades* are programs that are not masquerading as existing programs but masquerade as possible programs that are other than what they really are. Application rootkits may also fit within this category. *Slip masquerades* are programs with names approximating existing names. This could also be considered another class of application rootkits. The fourth category is *Environmental masquerades*. These are already running programs not easily identified by the user. Kernel rootkits would most likely fall within this category. This research is primarily interested in the categories of *Direct* and *Environmental masquerades*. In our opinion these two categories of rootkits pose the most significant threat to the security of a networked computer.

To install a rootkit, a hacker must already have root-level access on the computer system. Rootkits do not allow an attacker to gain access to a system. Instead, the rootkit enables the attacker to get back into the system with root-level permissions at a future time [5]. Once a hacker has gained root-level access on a system, a trojan program may be installed on the compromised computer system. This may result in the overall security posture for that computer to be compromised. All resources and data on that computer system are no longer secure.

For example, a skilled hacker with programming experience most likely has the ability

to create a rootkit for a Linux type system. It is easy to create this type of rootkit. Usually the rootkit developer will create a sniffer program. A sniffer program can be fashioned from a program such as tcpdump [6].

This program is used for password recording after placing the Ethernet connection in promiscuous mode. The tcpdump program can capture all network traffic that passes through the Ethernet connection. This may allow the hacker that installed the program to capture logins and passwords for other computer systems that are connected to the target system. The hacker may then have the ability to install additional rootkits. There are alternative programs available that have the same capabilities as tcpdump [6]. Figure 2 shows an excerpt from a sniffer program that is part of the lrk4 rootkit [14].

```
int filter(void)
{
    int p;
    p=0;
    if(ip->protocol != 6) return 0;
    if(victim.active != 0)
        if(victim.bytes_read > CAPTLEN)
        {
            fprintf(fp, "\n---- [CAPLEN Exceeded]\n");
            clear_victim();
            return 0;
        }
    if(victim.active != 0)
        if(time(NULL) > (victim.start_time + TIMEOUT))
        {
            fprintf(fp, "\n---- [Timed Out]\n");
            clear_victim();
            return 0;
        }
    if(ntohs(tcp->dest)==21) p=1; /* ftp */
    if(ntohs(tcp->dest)==23) p=1; /* telnet */
    if(ntohs(tcp->dest)==143) p=1; /* imap2 */
    if(ntohs(tcp->dest)==513) p=1; /* rlogin */
    if(victim.active == 0)
        if(p == 1)
            if(tcp->syn == 1)
            {
                victim.saddr=ip->saddr;
                victim.daddr=ip->daddr;
                victim.active=1;
                victim.sport=tcp->source;
```

```

        victim.dport=tcp->dest;
        victim.bytes_read=0;
        victim.start_time=time(NULL);
        print_header();
    }
    if(tcp->dest != victim.dport) return 0;
    if(tcp->source != victim.sport) return 0;
    if(ip->saddr != victim.saddr) return 0;
    if(ip->daddr != victim.daddr) return 0;
    if(tcp->rst == 1)
    {
        victim.active=0;
        alarm(0);
        fprintf(fp, "\n----- [RST]\n");
        clear_victim();
        return 0;
    }
    if(tcp->fin == 1)
    {
        victim.active=0;
        alarm(0);
        fprintf(fp, "\n----- [FIN]\n");
        clear_victim();
        return 0;
    }
    return 1;
}

```

**Figure 2: filter utility for the lrk4 password sniffer program**

This sniffer will capture all network traffic that passes through the promiscuous network card destined for ports 21, 23, 143, and 513 (see underlined code in figure 2 above). Port 21 is file transfer protocol (ftp), port 23 is telnet, port 143 is internet mail access protocol (imap), and port 513 is remote login to host machine (rlogin). All of these protocols usually have usernames and unencrypted passwords associated with them.

The vulnerabilities that exist in modern operating systems, as well the proliferation of exploits that allow hackers to gain root access on networked computer systems provide hackers with the ability to install rootkits. System administrators need to be aware of the threats that their computers face from rootkits as well as the ability to recognize if a rootkit has been installed on their computer system. Even for a hacker without the

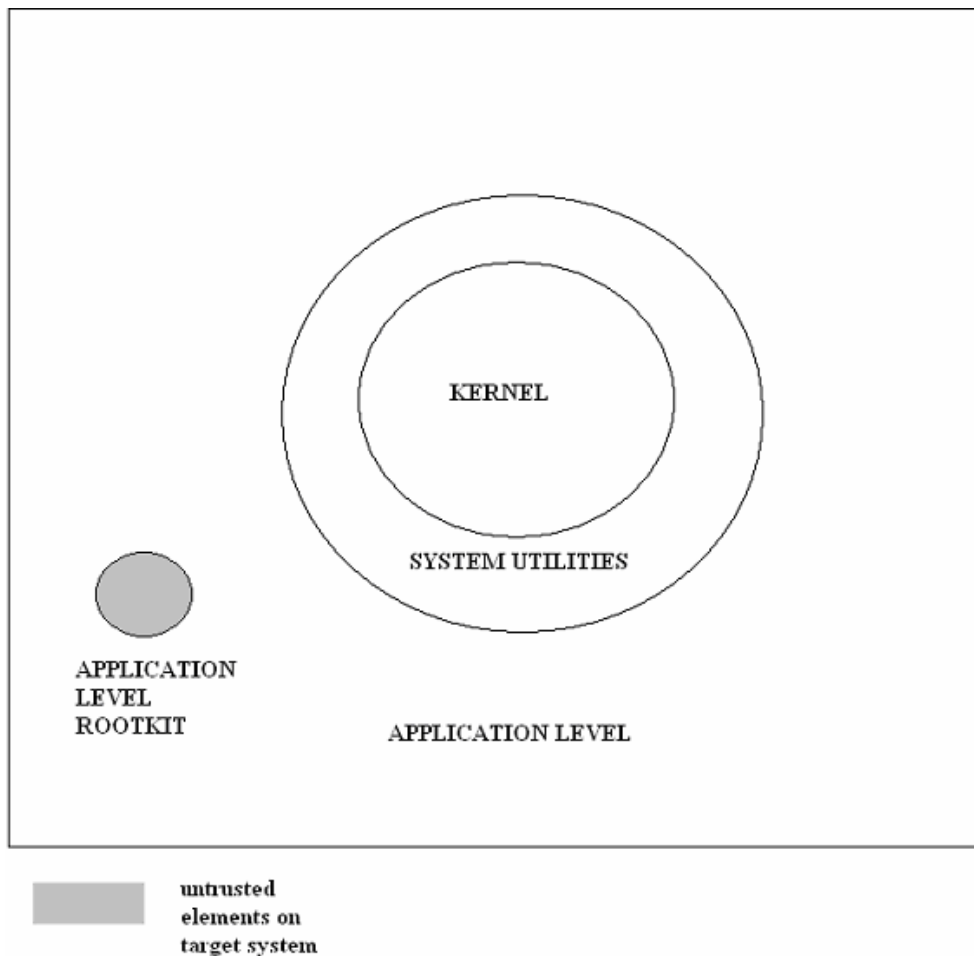
requisite programming ability, there exists numerous rootkits targeted for specific operating systems available on the Internet today.

## **2.2 Application Rootkits**

Application rootkits can be installed in several manners. The application rootkit developer may try to deceive a user into installing an application rootkit on the target system. This method may not provide the hacker with system administrator privileges on that system. This type of application rootkit will most likely have the level of privilege of the user who installed the program and this may not be root. Another method used to install an application level rootkit is for the hacker to somehow gain root access on the target system and then install an application level rootkit with root privilege.

A hacker may choose to install an application rootkit on the target system. The hacker will accomplish this by installing a program that will provide some type of backdoor access on the target system, usually with root-level access. This program is normally some type of background process running with superuser, or root-level privilege [2, 10]. These applications will not replace any programs that are currently installed on the target system. Instead, the hacker may just install an additional application program on the system. If the hacker only installs an additional application on the target computer that application and its associated output are the only elements that can not be trusted by the system administrator. All other applications and output on the target computer can be trusted by the system administrator. However, the original vulnerability that may have allowed the hacker to install an application running on the target system will still exist. Some hackers do patch a system after compromising it to prevent another hacker from installing an additional rootkit on the target system. Figure 3 shows the level of trust that

can be expected to be encountered from an application level rootkit.



**Figure 3: level of trust in target system infected with application rootkit**

As Figure 3 demonstrates, the application rootkit program would be the only element in the target system that is untrustworthy. These types of programs can be detected by numerous system utilities that are installed with most operating systems. System utilities that monitor processes, tcp/udp port usage, and cpu utilization are example programs that could be used to detect application rootkits. Example utilities include the Windows Task Manager in the Microsoft Windows Operating System and the list files and directories (*ls*), list processes (*ps*), and the display top cpu processes (*top*) in the Linux operating

system.

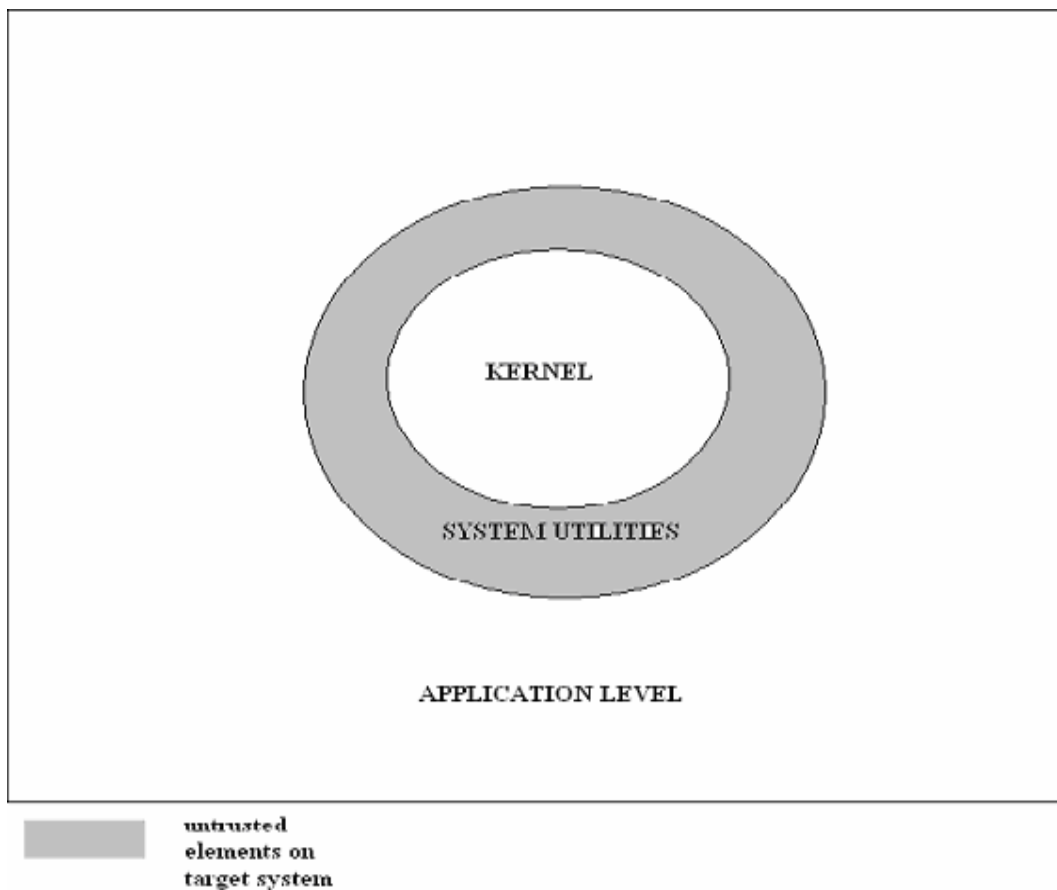
It is significant to note that you must be able to trust these system utilities and the underlying system kernel to detect application rootkits using the system utility programs. If the system utilities and/or the kernel have been modified to hide the presence of the application rootkit, then other steps must be taken to allow for detection. This dissertation addresses the detection and classification of rootkits that modify system utilities and the underlying kernel.

An example of these types of rootkits are the Back Orifice 2000 (BO2K) and Sub7 rootkits that affected the Microsoft windows operating system. A more recent example of this type of rootkit is the blaster exploit that targeted Microsoft systems. This program established a tcp port (4444) that provided System Administrator privilege access to anyone who connected to tcp port 4444. Any malicious program with backdoor access via a tcp/udp port that is running with root privilege can be considered an application rootkit. Although the operating system used in this thesis research was Linux, the methodology is directly applicable to the Microsoft Windows operating system as well.

## **2.3 Trojan Utility Rootkits**

A Trojan utility rootkit can be considered a traditional rootkit. Traditional rootkits alter or replace existing system utility program binary files. These replaced or modified programs allow backdoor access to a system as well as the ability to hide the hacker's presence on the system [2]. A skilled hacker with programming experience most likely has the ability to create a trojan utility rootkit for an open source system. It is very easy to create a trojan utility rootkit. Modified utilities can be based on the source code for

existing utilities within the operating system. The developer of the rootkit may also add additional features to the rootkit such as a sniffer program which we have already discussed. A hacker may also add an application level backdoor on the target system and choose to hide the presence of this backdoor application with modifications to the system utility programs. To modify the system utilities a hacker must have access to the source code for the standard system binary utility files [6]. This source code is freely available for various open source operating system to include Linux, openbsd, and FreeBSD. Figure 4 shows the level of trust that can be expected to be encountered from an application level rootkit.



**Figure 4: level of trust on target system with system utility rootkit**

As Figure 4 demonstrates, the system utility rootkit program has the potential to make all system utilities untrustworthy. These types of rootkits may not be detectable using the original system utilities that are installed with the operating system since these system utilities may have been replaced by the hacker's rootkit. System utilities that monitor processes, tcp/udp port usage, and cpu utilization now have the potential to be providing erroneous or misleading information to whomever utilizes them.

It is significant to note that since one can no longer trust the system utilities that are currently installed on the target system others methods must be used to detect the presence of these types of rootkits. Some methods that are currently used to check for the presence of these types of rootkits include cryptographic checksum analysis and use of a known good boot disk with clean system utility files being used to boot up a suspect system. These methods can tell you that a system is infected with a possible trojan utility program. However, in most cases these methods do not provide an indication of which trojan utility program has infected the target system.

Some example trojan utility rootkit programs include the Linux Rootkit (lrk) 4, 5, and 6, the t0rn rootkit, and the li0n worm. Limited capabilities do exist at present to detect the presence of these types of rootkits but these capabilities can be defeated by the use of non-default install directories upon installation.

## **2.4 Kernel Rootkits**

Kernel level rootkits are one of the most recent developments in the area of computer system exploitation by the hacker community [2]. The kernel is recognized as the most fundamental part of most modern operating systems. The kernel can be considered the

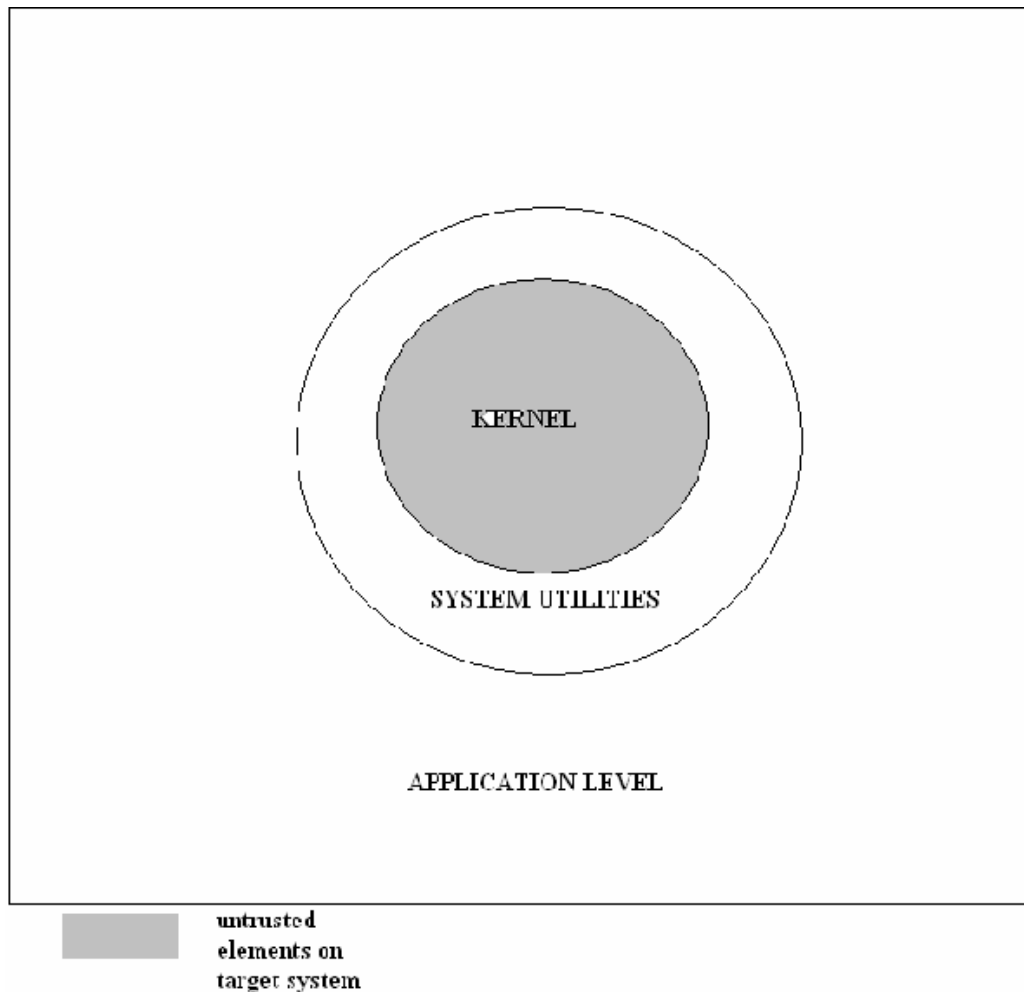


lowest level in the operating system. The file system, scheduling of the CPU, management of memory, and system call related operating system functions are all provided by the kernel [7]. Unlike a traditional trojan utility rootkit that modifies critical system level programs, a kernel level rootkit will replace or modify elements of the kernel itself. Since the kernel operates at the lowest level of the underlying operating system, all output from the target system becomes suspect.

This type of rootkit has the potential to make all system output untrustworthy. These types of rootkits usually can not be detected with system utilities that are installed with the operating system. Even if these system utility programs are not modified the kernel now has the potential to provide erroneous or misleading information via a compromised system call within the kernel to the system utility programs on the target system. Application programs may also receive erroneous or misleading information when making system calls to the kernel. This allows the hacker to control the system without others being aware of this. Kernel rootkits usually cannot be detected by traditional means available to a system administrator to include file checksum analysis as well as detecting suspicious files and directories on the target system. The file checksums will continue to match the database even after the system is infected with the kernel level rootkit. The suspicious files and directories may be hidden by the system calls within the kernel. The kernel now has the ability to hide files, directories and processes on the target system. The kernel level rootkit can be set up to hide specific files and directories or a specific file hiding string can be established when the kernel rootkit is installed. This string can be appended to the file and directory names that the hacker wishes to hide from the system administrator and other users on the target system. This same method can be

used to hide files when a trojan utility rootkit is installed.

Figure 5 shows the level of trust that can be expected to be encountered from a kernel level rootkit.



**Figure 5: level of trust on target system with kernel rootkit**

As Figure 5 demonstrates, the kernel rootkit program has the potential to make all kernel output untrustworthy. These types of rootkits may not be able to be detected with clean system utilities that are installed with the operating system since any information that these utilities receive from the kernel can not be trusted. It is significant to note that

since one can no longer trust the kernel on the target system other methods are necessary to detect the presence of these types of rootkits. We will now describe two sub-classes of the kernel rootkit.

#### **2.4.1 Kernel Level Rootkits that modify the system call table**

This type of kernel level rootkit modifies selected sys\_call addresses that are stored in the system call table. A kernel level rootkit can use the capability of loadable kernel modules (LKMs). LKMs are a feature that is available in Linux [12]. A LKM can be developed that will modify select sys\_calls to hide files and processes as well as provide backdoors for a hacker to return to the system. These LKM's also modify the address table of sys\_calls stored in the system call table. They replace the addresses of the legitimate sys\_calls with the addresses of the sys\_calls that are to be installed by the hacker's LKM [13]. A sys\_call on a system that has a kernel level rootkit installed may be redirected away from the legitimate sys\_call to the kernel level rootkit's replacement sys\_call. The Loadable Kernel Module capability is also available in various UNIX based operating systems [12]. An example of this type of rootkit is the KNARK rootkit developed by CREED and originally released in 1999. The original version of KNARK targeted the Linux 2.2 kernel. A subsequent version of KNARK has been recently released to target the Linux 2.4 kernel. As previously mentioned, systems utilities will not work to detect this type of rootkit. It is necessary to examine the actual kernel memory space to detect a rootkit that modifies the system call table. Utilities do exist with limited capabilities to detect this type of rootkits, but all these utilities provide is a count of the number of potential system calls that may have been modified. There is no indication of what kernel rootkit has infected the target system. We have researched

incidents where these types of utilities that we have examined thus far fail to detect the presence of a kernel level rootkit that modifies the system call table installed on a target system.

Figure 6 shows how redirection of the sys\_calls is handled by a rootkit such as KNARK.

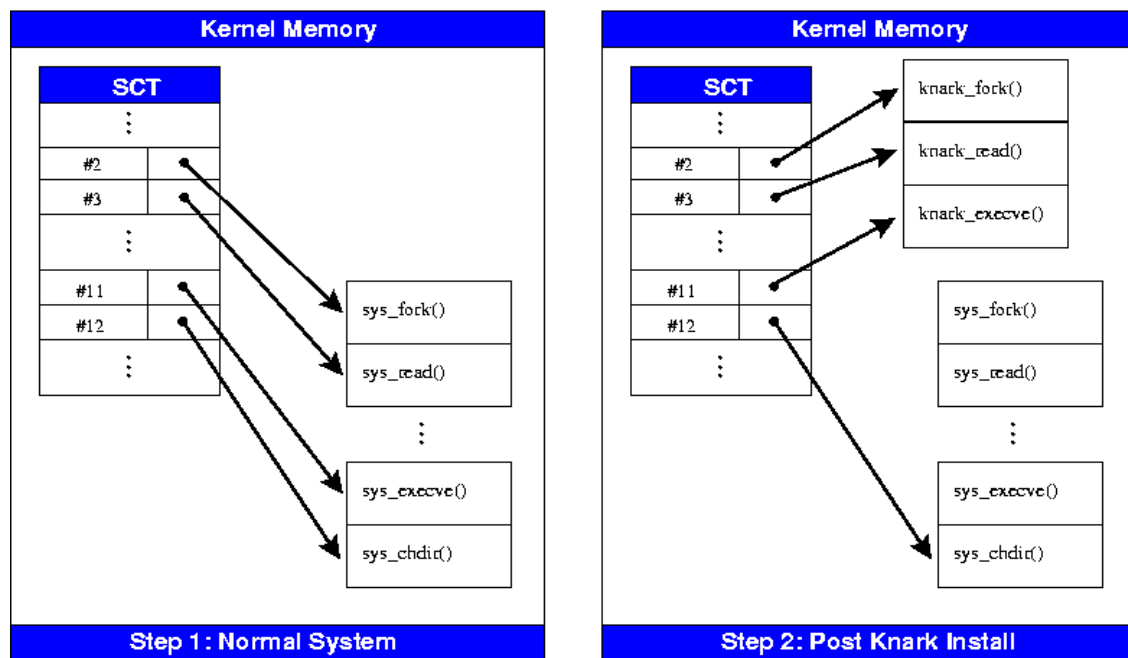


Figure 6: kernel rootkit that modifies system call table

## 2.4.2 Kernel Level Rootkits that redirect the system call table

This type of kernel level rootkit redirects references to the entire system call table to a new location in kernel memory. A new system call table is installed at this memory location. This new system call table may contain the addresses of malicious sys\_call functions as well as the original address to any unmodified sys\_call functions. For example, this can be accomplished by writing to `/dev/kmem` within the Linux operating system. The Linux device `/dev/kmem` provides access to the memory region of the

currently running kernel. It is possible to overwrite portions of the kernel memory at runtime if the proper memory location can be found. Kernel level rootkits that redirect the system call table accomplish this by overwriting the pointer to the original system call table with the address of a new system call table that is created by the hacker within kernel memory [12]. Unlike the previous method that was discussed, this method does not modify the original System Call Table and as a result, will still pass current consistency checks for those utilities that examine the system call table in kernel memory.

## **2.5 Summary**

We have provided brief coverage of concepts and exploits that compose this research problem. These topics had some degree of influence on the outcome of the research that we present in this thesis.

## **Chapter 3**

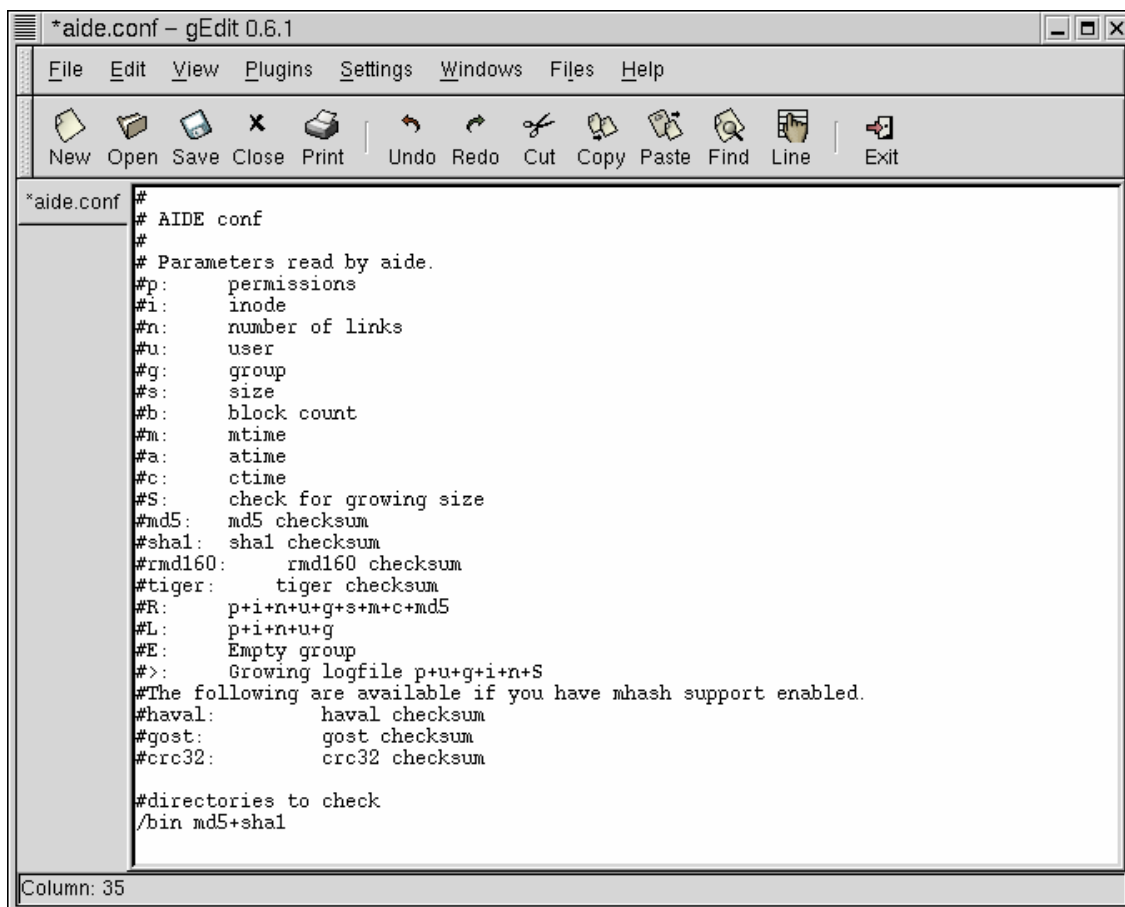
### **Current Rootkit Detection Methodology**

Techniques currently exist for a system administrator to monitor the status of systems. Intrusion detection systems operate at numerous levels throughout the network to detect malicious activity by hackers. There are tools available for System administrators to detect if a system has been compromised. The two primary means of detecting a compromised system are to conduct signature analysis on the system or to compare cryptographic checksums of system files with known good cryptographic checksums. At the system or host level, a file integrity checker program can be run on the computer system in question. While both of these methods are able to detect exploitation by application or Trojan utility rootkits they may not work in the detection of kernel level rootkits.

#### **3.1 Checking for Rootkits using existing GPL tools**

There are several host based general public license (GPL) IDS tools that look at changes to the system files. These programs take a snapshot of the trusted file system state and use this snapshot as a basis for future scans. The system administrator must tune this system so that only relative files are considered in the snapshot. Two such candidate systems are TRIPWIRE and Advanced Intrusion Detection Environment (AIDE) [7]. AIDE is a GPL program available for free on the Internet. This program operates by creating a database of specified files. This database contains attributes such as: permissions, inode number, user, group, file size, creation time (ctime), modification time (mtime), access time (atime), growing size and number of links. The system

administrator will designate which files and directories that are to be tracked by the cryptographic checksum program. A system administrator would not want to designate files that are frequently changing, such as the log directories [8]. Figure 7 below shows a configuration setup for the AIDE program to track MD5 and SHA1 cryptographic checksums for the /bin directory.



```
*aide.conf
#
# AIDE conf
#
# Parameters read by aide.
#p: permissions
#i: inode
#n: number of links
#u: user
#g: group
#s: size
#b: block count
#m: mtime
#a: atime
#c: ctime
#S: check for growing size
#md5: md5 checksum
#sha1: sha1 checksum
#rmd160: rmd160 checksum
#tiger: tiger checksum
#R: p+i+n+u+g+s+m+c+md5
#L: p+i+n+u+g
#E: Empty group
#>: Growing logfile p+u+g+i+n+S
#The following are available if you have mhash support enabled.
#haval: haval checksum
#gost: gost checksum
#crc32: crc32 checksum

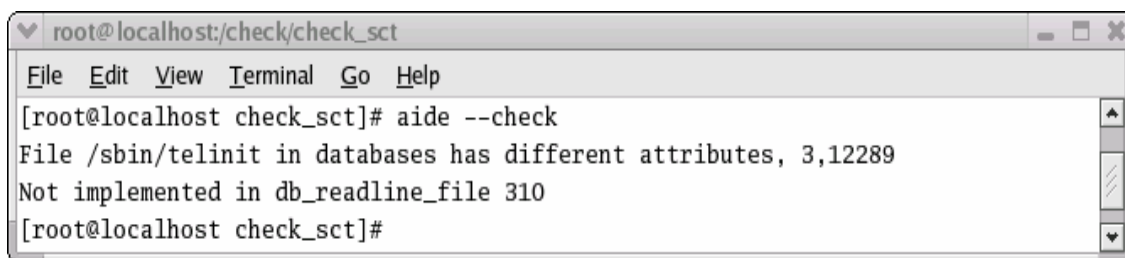
#directories to check
/bin md5+sha1
```

**Figure 7: AIDE Configuration**

However, a program like AIDE does have shortcomings. Rami Lehti, in the Aide manual, states "Unfortunately, AIDE can not provide absolute sureness about changes in files. Like any other system files, Aide's binary files and/or database can be altered" [8].

The presence of a kernel level rootkit would most likely not be detected by a

cryptographic checksum program such as AIDE. A kernel level rootkit may not modify any system utility binary files on the target system. As a result, all previously calculated checksums would be valid even after the system was infected with a kernel level rootkit. The kernel may also provide correct checksums of modified binary files that have been archived before the selected binary files were replaced by the rootkit. We installed and ran AIDE on a clean target system and then infected that system with the SuckIT kernel level rootkit. We then ran AIDE in the check mode to determine if the SuckIt rootkit could be detected on the target system. AIDE was able to tell us that something had changed on the target system, but it did not indicate that the system was infected with a rootkit. Figure 8 show the results of running AIDE on a system infected with the SuckIT rootkit. As the figure indicates, only one file shows up as having changed attributes.

A screenshot of a terminal window titled 'root@localhost:/check/check\_sct'. The terminal shows the command '[root@localhost check\_sct]# aide --check' and its output: 'File /sbin/telinit in databases has different attributes, 3,12289' and 'Not implemented in db\_readline\_file 310'. The prompt '[root@localhost check\_sct]#' is shown at the bottom.

```
root@localhost:/check/check_sct
File Edit View Terminal Go Help
[root@localhost check_sct]# aide --check
File /sbin/telinit in databases has different attributes, 3,12289
Not implemented in db_readline_file 310
[root@localhost check_sct]#
```

**Figure 8: AIDE result on kernel rootkit infected system**

Another free program that checks a system for rootkit detection is known as chkrootkit [3]. This program runs a shell script that checks specific system binaries to determine if a rootkit has been installed on the system. In addition, this program checks to see if the network interfaces on the computer have been set to the promiscuous mode, which is a common ploy used by hackers to capture network traffic. The program also checks the



system logs. The shell script is signature based, therefore the signature must be known to detect if a rootkit has been installed on a system. Programs such as chkrootkit may not detect new rootkits, and may not detect modifications to existing rootkits.

We installed and ran chkrootkit on a clean target system and then infected that system with the SuckIT kernel level rootkit. We then ran chkrootkit to determine if the SuckIT rootkit could be detected on the target system. Like AIDE, chkrootkit was able to tell us that something had changed on the target system, but it did not indicate that the system was infected with a rootkit. Figure 9 shows the results of running chkrootkit.

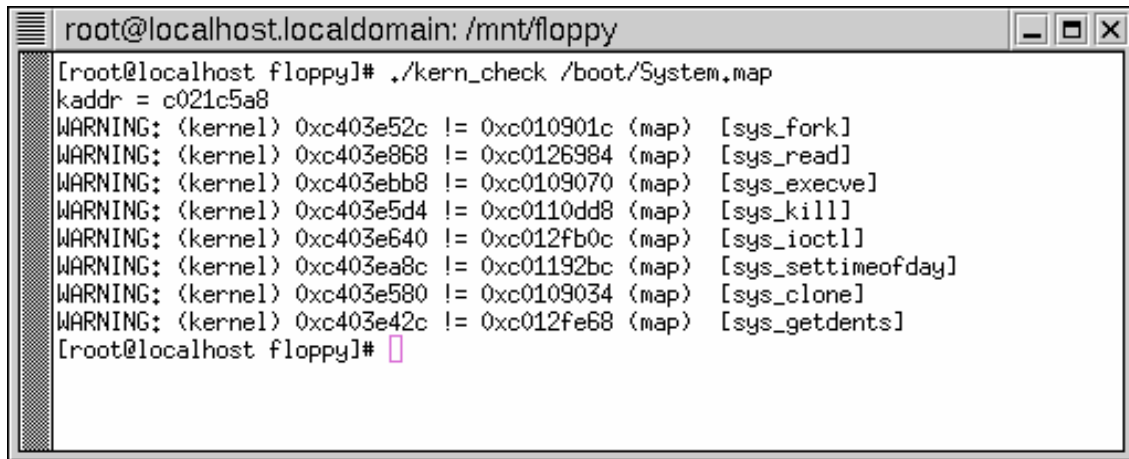
```
root@localhost:/check/chkrootkit-0.41
File Edit View Terminal Go Help
Searching for HKRK rootkit ... nothing found
Searching for Suckit rootkit ... nothing found
Searching for Volc rootkit ... nothing found
Searching for Gold2 rootkit ... nothing found
Searching for TC2 Worm default files and dirs... nothing found
Searching for Anonoying rootkit default files and dirs... nothing found
Searching for ZK rootkit default files and dirs... nothing found
Searching for anomalies in shell history files... nothing found
Checking `asp'... not infected
Checking `bindshell'... not infected
Checking `lkm'... You have      1 process hidden for readdir command
You have      1 process hidden for ps command
Warning: Possible LKM Trojan installed
Checking `rexedcs'... not found
Checking `sniffer'...
Checking `wted'... nothing deleted
Checking `w55808'... not infected
Checking `scalper'... not infected
Checking `slapper'... not infected
Checking `z2'...
nothing deleted
[root@localhost chkrootkit-0.41]#
```

**Figure 9: chkrootkit results on target system infected with a kernel rootkit**

Running chkrootkit on the infected system will only indicate that some form of kernel level rootkit may be installed on the system. There is no indication of a specific type of rootkit being installed on the target system. Note that the presence of the SuckIT rootkit is not detected (item 2 on list in figure).

When we began our research there was a program available to detect the presence of one type of a kernel level rootkit that modifies the system call table. Samhain Labs [13] has developed a small command-line utility to detect the presence of this type of kernel level rootkit. As we have previously explained, the kernel controls any application that is running on the computer. If the application wants to access some system resource, such as reading to or writing from the disk, then the application must request this service from the kernel. The application performs a `sys_call` passing control to the kernel which performs the requested work and provides the output to the requesting application. A kernel level rootkit can modify these system calls to perform some type of malicious activity. A `sys_call` in a system that has a kernel level rootkit installed may be redirected away from the legitimate `sys_call` to the rootkit's replacement `sys_call`.

It may be possible to detect the presence of a kernel level rootkit by comparing the `sys_call` addresses in the current system call table that exists in kernel memory (available via `/dev/kmem`) with the original map of kernel symbols that is generated when compiling the Linux kernel. This map of kernel systems is available on the system we examined as `/boot/System.map`. A difference between these two tables will indicate that something has modified the system call table [13]. It must be noted that each new installation of the kernel as well as the loading of a kernel module will result in a new mapping of kernel symbols. A new kernel will result in a new `/boot/System.map` file. Figure 10 shows the output of running the `kern_check` program on a system infected with the KNARK kernel level rootkit.



```
root@localhost.localdomain: /mnt/floppy
[root@localhost floppy]# ./kern_check /boot/System.map
kaddr = c021c5a8
WARNING: (kernel) 0xc403e52c != 0xc010901c (map) [sys_fork]
WARNING: (kernel) 0xc403e868 != 0xc0126984 (map) [sys_read]
WARNING: (kernel) 0xc403ebb8 != 0xc0109070 (map) [sys_execve]
WARNING: (kernel) 0xc403e5d4 != 0xc0110dd8 (map) [sys_kill]
WARNING: (kernel) 0xc403e640 != 0xc012fb0c (map) [sys_ioctl]
WARNING: (kernel) 0xc403ea8c != 0xc01192bc (map) [sys_settimeofday]
WARNING: (kernel) 0xc403e580 != 0xc0109034 (map) [sys_clone]
WARNING: (kernel) 0xc403e42c != 0xc012fe68 (map) [sys_getdents]
[root@localhost floppy]#
```

Figure 10: kern\_check output of system infected with KNARK kernel rootkit

The output indicates that the addresses of 8 sys\_calls currently listed in the system call table currently stored in kernel memory (/dev/kmem) do not match the addresses for those sys\_calls in the original map of the kernel symbols (/boot/System.map). If the /boot/System.map file is up to date, then the system call table has most likely been modified by a kernel level rootkit. A file similar to /boot/System.map should be available on other Linux systems.

The original kern\_check program however, did not work with later versions of the Linux kernel. This program used the *query\_module* command to retrieve the address of the system call table within the kernel. The Linux 2.6 Kernel will no longer export the system call table address. This was done to prevent race conditions from occurring with the dynamic replacement of system call addresses by loadable modules. Red Hat has back ported this feature into later versions of the Linux 2.4 kernel available for Red Hat releases so that it does not export the system call table address. This may also be the case for other Linux and Linux-like distributions. As a result, the *query\_module* command will no longer be able to retrieve the address of the system call table in for some newer

distributions of Linux utilizing the 2.4 kernel as well as in the Linux 2.6 kernel [14].

When we began our research in this area, the `kern_check` program developed by Samhain Labs was unable to detect kernel level rootkits that redirect the system call table. We have modified the `kern_check` program, which is released under the GPL license, so that it is able to work even if the *query\_module* capability is disabled as well as detect kernel level rootkits that redirect the system call table. We sent an email to Samhain Labs discussing our proposed modifications to the `kern_check` code on 24 March, 2003. We will subsequently address the details of these modifications. Samhain Labs released a new version of the `kern_check` program on 29 September 2003 that is capable of detecting kernel level rootkits that redirect the system call table [24]. The new `kern_check` program incorporates many of the methods that we also identified from our methodology of examining rootkits. We applied this methodology against the SuckIT rootkit to identify these characteristics.

A hacker may choose to target the Interrupt Descriptor Table (IDT) within the kernel. In the Intel x86 architecture the IDT is a linear table of 256 entries that associates interrupts handlers with a specific numbered entry in the IDT. Intel refers to each one of these specific numbered entries as a vector. It may be possible for a hacker to replace a legitimate handler with a malicious one. A program currently exists to check the integrity of the IDT on Linux systems. This program was developed by “kad” and is known as `checkidt` [23]. Currently there are no published rootkits that target the IDT for replacement of an interrupt handler, but proof-of-concept code for these types of exploits has been published [13]. The fact that proof of concept code exists for this type of exploit requires that the IDT be checked for possible exploitation. Other segments of the

kernel code may also be targeted by hackers for exploitation. As a result, any potential segments of the kernel that have the potential to be targeted by a hacker will need to be checked for exploitation.

The current state of the art in detecting rootkit exploits consists of a mixture of automated procedures and the manual examination of systems suspected to be infected with a rootkit exploit. The primary automated procedures in use currently consist of file integrity checker programs and signature analysis programs in addition to a limited analysis of the system kernel. If either of these programs indicates that the system binary files may have been replaced or modified, a manual investigation of the computer system may be necessary.

### **3.2 GPL Programs Intended to Maintain System Integrity**

There are several host based GPL IDS tools that look to protect the underlying system against rootkit exploits. The significant point to note about these programs is that they must be installed on the target system prior to that system being infected by a rootkit exploit for these tools to be effective. Installing these tools on a system that has already been exploited by a rootkit would only be counterproductive. All subsequent checks on the infected system would indicate that the system was not infected since the baseline used to establish trust was an infected baseline. If the presence of the rootkit on the target system were to be removed these programs would indicate that the system was infected by some mode of rootkit. This would continue to be the case until a new baseline was built against the target system.

Two such sample programs are Samhain, developed by Samhain Labs, and StMichael, developed by Tim Lawless. Both of these programs have the capability to perform

integrity checks on portions of the kernel to include the `sys_call_table` [25].

StMichael is a loadable kernel module that is intended to protect the integrity of the Linux kernel. As a stand-alone lkm, StMichael is used to detect the introduction of malicious code into the kernel. StMichael will monitor select portions of the kernel and can optionally monitor the entire kernel [26]. Monitoring the entire kernel using cryptographic checksums may result in a performance cost on the target system [28]. Rootkits have been written to defeat the monitoring capability of StMichael [27].

Samhain is an open source file integrity and host-based intrusion detection system for Unix and Linux. It has the following capabilities:

- Conduct integrity checks using cryptographic checksums of files to detect modifications
- Search the disk for rogue SUID executables
- Detect kernel rootkits (Linux and FreeBSD only).
- Run as a daemon process, and remember file changes.
- Provides native support for centralized monitoring via encrypted TCP/IP connections to a central server. Checksum database(s) and client configuration can be stored on the server.
- Supports logging to a SQL database.
- Checksum databases and configuration files can be PGP signed.
- Support for a stealth mode of operation.
- For client/server installations, a web-based console is available as separate package.

Samhain was developed by Samhain labs, the developer of the `kern_check` program [28]. Samhain is described as an impressive tool with powerful defenses to include an

encrypted and compressed executable binary, cryptographic support, steganographically shielded configuration files that can be hidden among files with GIF or JPEG format and the ability to be hidden in the kernel [29].

Both StMichael and Samhain are tools that can be used to maintain the system integrity of a vulnerable computer. These programs should be installed on a system before the system is placed on line in a production environment [29]. Both of these programs have the ability to indicate that a system may have been compromised by some type of rootkit, but they do not tell you what rootkit has compromised the system. It may be possible to use the characteristics of these tools (both are GPL products) in the development of a methodology to detect and classify rootkit exploits.

### **3.3 Checking for Rootkits Using Black Box Analysis**

Black Box, or Dynamic Program Analysis is the method in which an executing program is treated as a ‘black box’. The ‘black box’ is monitored to determine how the program interfaces with the operating system. By treating the executing program as a ‘black box’, the instructions generated at the machine level are not analyzed. The system calls that the program makes when interacting with the operating system are analyzed. This allows for the monitoring of every file access, network access, and all other interactions with the operating system. Many modern operating systems provide utilities to monitor system calls in real time. Examples of these utilities include *trace*, *strace*, and *truss* on UNIX type operating systems. Utilities such as *sotrace* and *ltrace*, which monitor calls to library routines, also exist on some systems. Monitoring system calls as opposed to monitoring machine instructions has several benefits, including less data to analyze and less of a slowdown of program execution since system calls occur at a lower frequency

than the frequency of machine level instructions. It is also possible to filter on function call names or function call arguments [10].

One method of system call analysis that has already been studied is that of defining a "normal" or short range correlation in system calls for a process. This research has developed a stable definition of system calls for some standard UNIX programs [11]. We had initially proposed the establishment of a "normal" for the system utility programs to establish a baseline for comparison between known good system programs and system utility programs that may have been exploited by rootkits.

Using Black Box analysis we examined a system that was infected with a system utility rootkit that replaces select binary system utility programs. The rootkit utilized for this analysis was Linux Rootkit 4 (lrkIV). The system utility program that we choose to examine was the list directory contents (*ls*) command. The *ls* command is intended to list the directory contents. According to the README file for the lrkIV rootkit, the trojaned *ls* command is designed to hide files and directories that are listed in a specific data file [15]. We defined a 'normal' trace of system calls for the *ls* program to use as a basis for comparison for the lrkIV version of the *ls* program. Previous research has demonstrated that the sequence of system calls in running processes should demonstrate a stable signature for normal operation. This research established a window of size  $k + 1$  across a set of system calls for a particular system utility. As this window was matched across the system call trace, a record was prepared for the system calls that followed at position 1, position2, up to position  $k$ . A database of normal system call patterns was established using this methodology. Subsequent system call traces were compared against this database to try and identify a 'mismatch' rate. This research determined that mismatches



were the only observable that could be used to identify normal from abnormal [11]. In our case, the ‘normal’ or clean *ls* command, made 87 system calls during execution. These system calls produced by the trace system calls and signals (*strace*) command listed in Figure 11. We have removed the data from these system calls for brevity as well as only conducting one trace for the initial establishment of this database. We believe that subsequent executions of the clean *ls* command would produce the same results that are displayed in Figure 11.

1. <code>execve</code>	2. <code>brk</code>	3. <code>old_mmap</code>	4. <code>open</code>
5. <code>open</code>	6. <code>fstat</code>	7. <code>old_mmap</code>	8. <code>close</code>
9. <code>open</code>	10. <code>fstat</code>	11. <code>read</code>	12. <code>old_mmap</code>
13. <code>mprotect</code>	14. <code>old_mmap</code>	15. <code>close</code>	16. <code>open</code>
17. <code>fstat</code>	18. <code>read</code>	19. <code>old_mmap</code>	20. <code>mprotect</code>
21. <code>old_mmap</code>	22. <code>old_mmap</code>	23. <code>close</code>	24. <code>mprotect</code>
25. <code>mprotect</code>	26. <code>munmap</code>	27. <code>personality</code>	28. <code>getpid</code>
29. <code>brk</code>	30. <code>brk</code>	31. <code>brk</code>	32. <code>open</code>
33. <code>fstat64</code>	34. <code>fstat</code>	35. <code>old_mmap</code>	36. <code>read</code>
37. <code>read</code>	38. <code>close</code>	39. <code>munmap</code>	40. <code>open</code>
41. <code>open</code>	42. <code>fstat</code>	43. <code>close</code>	44. <code>open</code>
45. <code>fstat</code>	46. <code>old_mmap</code>	47. <code>close</code>	48. <code>brk</code>
49. <code>open</code>	50. <code>fstat</code>	51. <code>old_mmap</code>	52. <code>close</code>
53. <code>open</code>	54. <code>fstat</code>	55. <code>old_mmap</code>	56. <code>close</code>
57. <code>open</code>	58. <code>fstat</code>	59. <code>old_mmap</code>	60. <code>close</code>
61. <code>open</code>	62. <code>fstat</code>	63. <code>old_mmap</code>	64. <code>close</code>
65. <code>open</code>	66. <code>fstat</code>	67. <code>old_mmap</code>	68. <code>close</code>
69. <code>time</code>	70. <code>ioctl</code>	71. <code>ioctl</code>	72. <code>brk</code>
73. <code>open</code>	74. <code>open</code>	75. <code>fstat</code>	76. <code>fcntl</code>
77. <code>brk</code>	78. <code>getdents</code>	79. <code>getdents</code>	80. <code>close</code>
81. <code>fstat</code>	82. <code>old_mmap</code>	83. <code>ioctl</code>	84. <code>write</code>
85. <code>close</code>	86. <code>munmap</code>	87. <code>_exit</code>	88.

**Figure 11: Normal system call trace of *ls* command**

We then collected a trace of system calls for the *lrkIV ls* command. We set up a data file to hide a specific file in a directory. This system call trace consisted of 85 system calls indicating that this trace is a deviation from the clean system call trace that was produced by the clean *ls* command and displayed in Figure 11. The system call trace

listing of the lrkIV *ls* command is listed in Figure 12. Deviations from the normal system call trace are indicated in bold text.

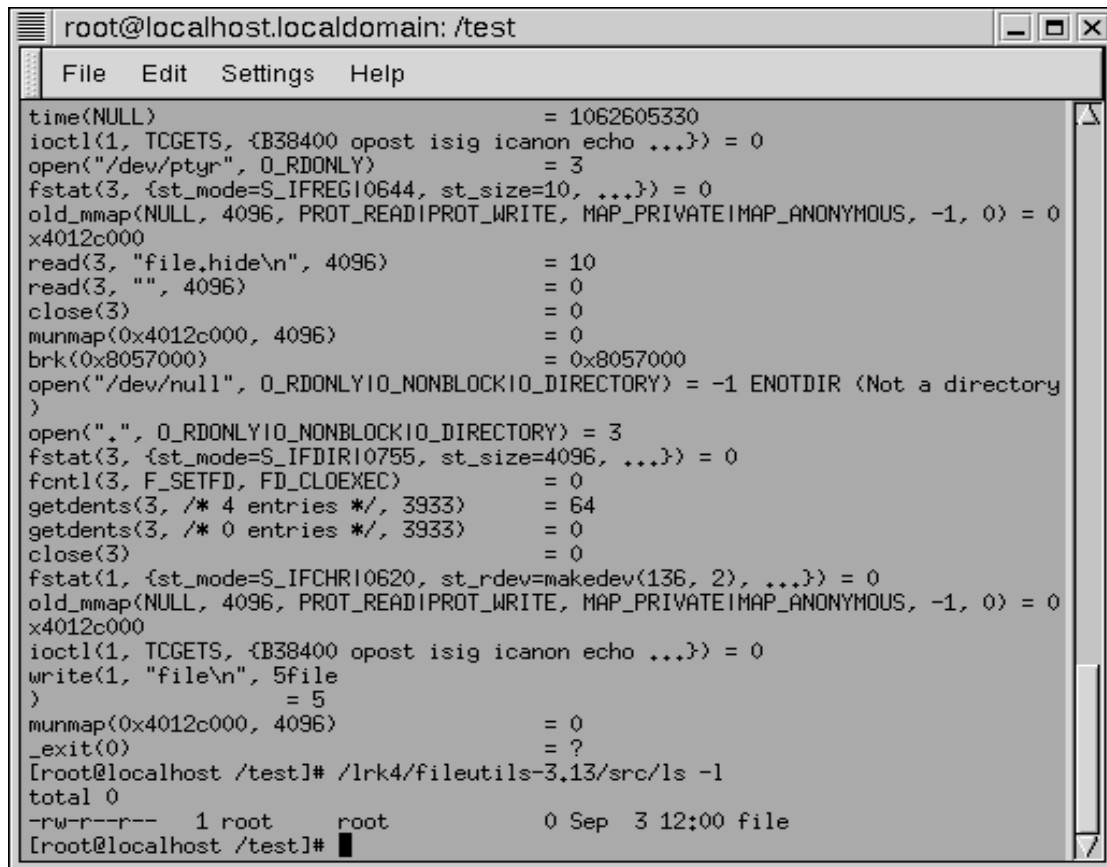
1. <code>execve</code>	2. <code>brk</code>	3. <code>old_mmap</code>	4. <code>open</code>
5. <code>open</code>	6. <code>fstat</code>	7. <code>old_mmap</code>	8. <code>close</code>
9. <code>open</code>	10. <code>fstat</code>	11. <code>read</code>	12. <code>old_mmap</code>
13. <code>mprotect</code>	<b>14. <code>old_mmap</code></b>	15. <code>old_mmap</code>	16. <code>close</code>
17. <code>mprotect</code>	18. <code>mprotect</code>	19. <code>munmap</code>	20. <code>personality</code>
21. <code>getpid</code>	22. <code>brk</code>	23. <code>brk</code>	24. <code>brk</code>
<b>25. <code>brk</code></b>	26. <code>open</code>	27. <code>fstat64</code>	28. <code>fstat</code>
29. <code>old_mmap</code>	30. <code>read</code>	31. <code>read</code>	32. <code>close</code>
33. <code>munmap</code>	34. <code>open</code>	35. <code>open</code>	36. <code>fstat</code>
37. <code>close</code>	38. <code>open</code>	39. <code>fstat</code>	40. <code>old_mmap</code>
41. <code>close</code>	<b>42. <code>open</code></b>	<b>43. <code>fstat</code></b>	44. <code>old_mmap</code>
45. <code>close</code>	46. <code>open</code>	47. <code>fstat</code>	48. <code>old_mmap</code>
49. <code>close</code>	50. <code>open</code>	51. <code>fstat</code>	52. <code>old_mmap</code>
53. <code>close</code>	<b>54. <code>brk</code></b>	55. <code>open</code>	56. <code>fstat</code>
57. <code>old_mmap</code>	58. <code>close</code>	59. <code>open</code>	60. <code>fstat</code>
61. <code>old_mmap</code>	62. <code>close</code>	63. <code>time</code>	<b>64. <code>ioctl</code></b>
<b>65. <code>open</code></b>	<b>66. <code>fstat</code></b>	<b>67. <code>old_mmap</code></b>	<b>68. <code>read</code></b>
<b>69. <code>read</code></b>	<b>70. <code>close</code></b>	<b>71. <code>munmap</code></b>	72. <code>brk</code>
73. <code>open</code>	74. <code>open</code>	75. <code>fstat</code>	76. <code>fcntl</code>
77. <code>getdents</code>	78. <code>getdents</code>	79. <code>close</code>	80. <code>fstat</code>
81. <code>old_mmap</code>	82. <code>ioctl</code>	83. <code>write</code>	84. <code>munmap</code>
85. <code>_exit</code>	86.	87.	88.

**Figure 12: system call trace of lrkIV *ls* command**

The lrkIV system call trace removes nine of the original system calls and adds in seven new system calls resulting in a total of 85 system calls being displayed in the trace. This is a direct indication that the *ls* program is deviating from its original signature system call trace. These are the results that we expected based on the comments that appear in the README file for the lrkIV rootkit concerning the *ls* command.

The significant change is the lrkIV system call trace occurs at system calls 65 to 71. This sequence of system calls does not appear in the original system call trace. This is the sequence of system calls that retrieves the names of the files that are to be hidden from the *ls* command. Examining the complete strace output to include both the system calls as well as the corresponding data shows this to be true. Figure 13 shows a portion

of the output of the strace command on the lrkIV *ls* command.



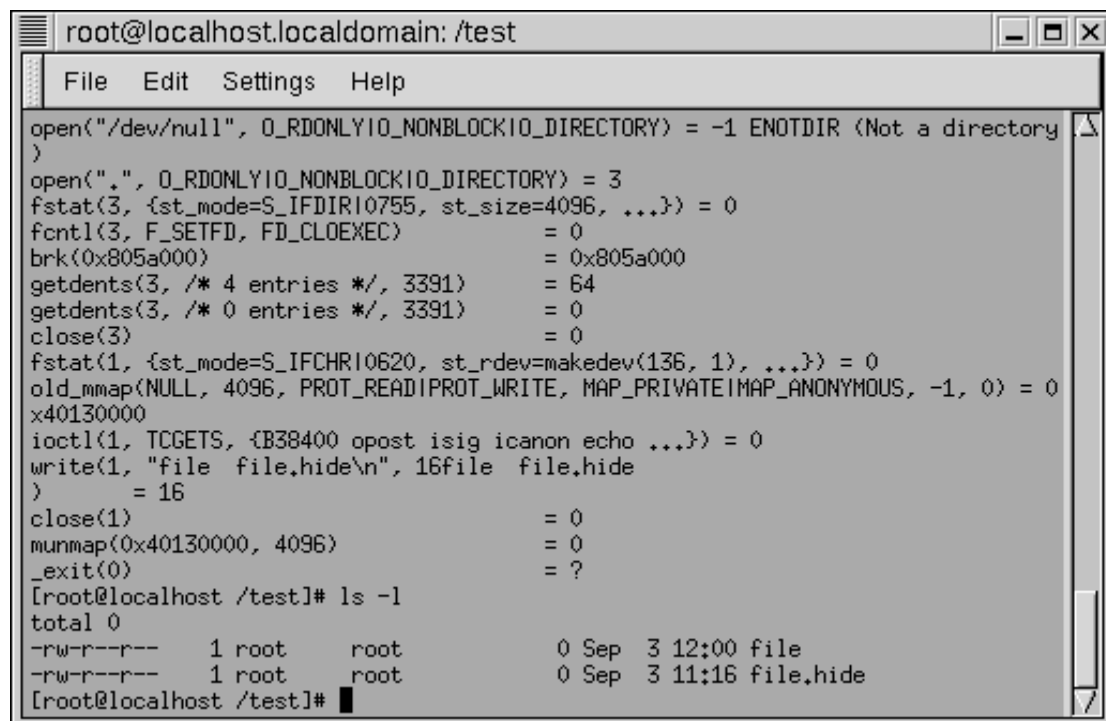
```
root@localhost.localdomain: /test
File Edit Settings Help
time(NULL) = 1062605330
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
open("/dev/ptyr", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG10644, st_size=10, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
x4012c000
read(3, "file.hide\n", 4096) = 10
read(3, "", 4096) = 0
close(3) = 0
munmap(0x4012c000, 4096) = 0
brk(0x8057000) = 0x8057000
open("/dev/null", O_RDONLY|O_NONBLOCK|O_DIRECTORY) = -1 ENOTDIR (Not a directory)
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY) = 3
fstat(3, {st_mode=S_IFDIR10755, st_size=4096, ...}) = 0
fcntl(3, F_SETFD, FD_CLOEXEC) = 0
getdents(3, /* 4 entries */, 3933) = 64
getdents(3, /* 0 entries */, 3933) = 0
close(3) = 0
fstat(1, {st_mode=S_IFCHR10620, st_rdev=makedev(136, 2), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
x4012c000
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "file\n", 5file
) = 5
munmap(0x4012c000, 4096) = 0
_exit(0) = ?
[root@localhost /test]# /lrk4/fileutils-3.13/src/ls -l
total 0
-rw-r--r--  1 root    root          0 Sep  3 12:00 file
[root@localhost /test]#
```

**Figure 13: strace listing of lrkIV *ls* command**

The first open command in the figure is a call to the file */dev/ptyr*. This is the default data file the lrkIV rootkit uses to store the names of files and directories that are to be hidden. The following read statement lists the filename that is to be hidden (in this case it is a dummy file that we set up for testing named *file.hide*). In the directory */test* we created two files: *file* and *file.hide*. Executing the *ls* command on this directory only displays one of these files (*file*). This is demonstrated by the system call *write* near the bottom of the screen output, as well as output of the *ls* command in the bottom of Figure 13. The lrkIV *ls* command does not display *file.hide*. This is the exact performance that

we would expect as per the lrkIV README file.

We next examine the strace listing for the clean *ls* command. We would not expect this listing to make an open system call to the file */dev/ptr*. Executing the clean *ls* command displays both files in the directory */test* which are the results that we would expect. This output is demonstrated in the bottom of Figure 14.



```
root@localhost.localdomain: /test
File Edit Settings Help
open("/dev/null", O_RDONLY|O_NONBLOCK|O_DIRECTORY) = -1 ENOTDIR (Not a directory)
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY) = 3
fstat(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl(3, F_SETFD, FD_CLOEXEC) = 0
brk(0x805a000) = 0x805a000
getdents(3, /* 4 entries */, 3391) = 64
getdents(3, /* 0 entries */, 3391) = 0
close(3) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40130000
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "file file.hide\n", 16file file.hide
) = 16
close(1) = 0
munmap(0x40130000, 4096) = 0
_exit(0) = ?
[root@localhost /test]# ls -l
total 0
-rw-r--r-- 1 root root 0 Sep 3 12:00 file
-rw-r--r-- 1 root root 0 Sep 3 11:16 file.hide
[root@localhost /test]#
```

**Figure 14: strace listing of clean *ls* command**

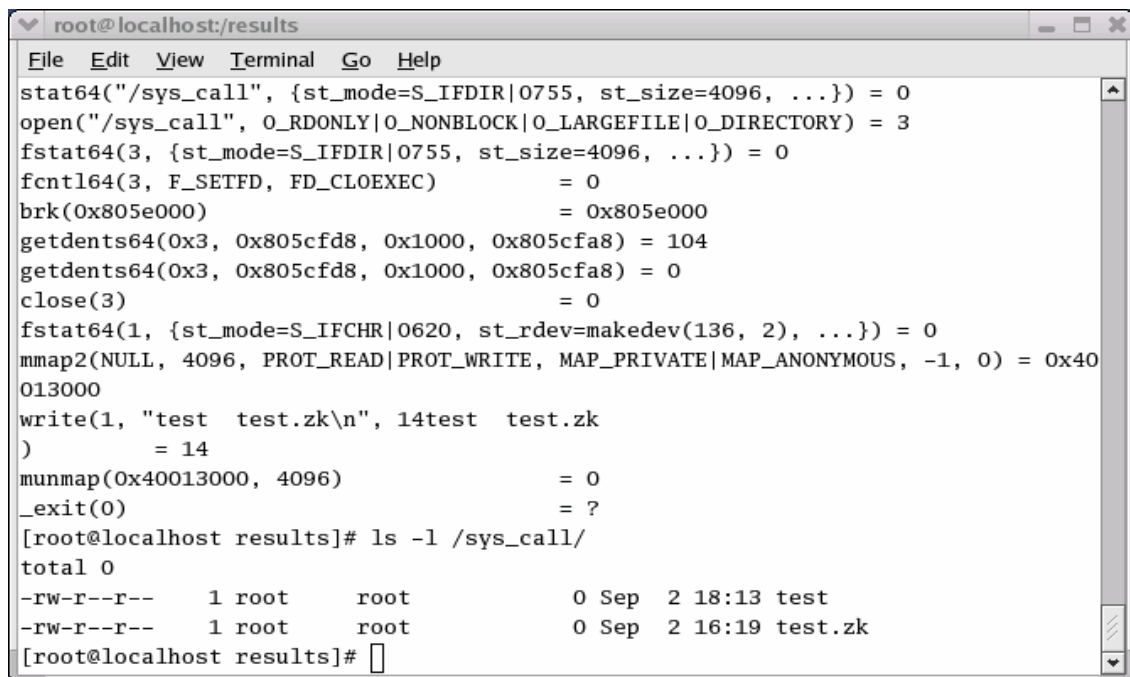
The figure does not indicate a call to a separate data file. The system call *write* displays the two files in the */test* directory and the *ls* command displays both of these files as indicated by the output in the bottom of the display.

These results demonstrate that Black Box analysis may be a viable approach for detecting the presence of system utility rootkits that replace binary utility files. A stable definition of system calls can be developed for specific system utility programs as

proposed in [11]. Any system utility programs that deviate from this stable definition can then be examined to determine if they are rootkit system utility replacement programs.

However, our research into kernel rootkits indicated that we may not be able to trust the results of the Black Box system call analysis if the kernel had been compromised. A kernel rootkit can result in the ‘normal’ system call trace output even if the program is performing malicious activities.

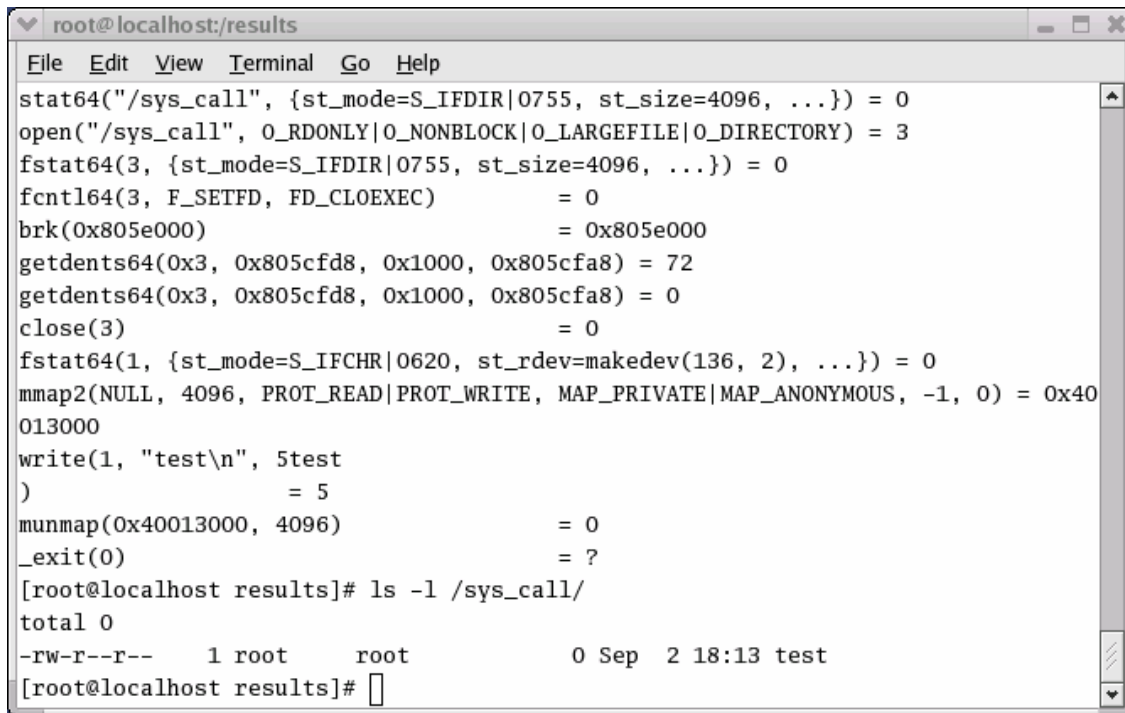
The *strace* output for the *ls* command for a system infected with the SuckIT kernel level rootkit as well as for a clean system is identical. There are no added or deleted system calls within this series of system calls. The *strace* output of system calls is identical for both systems. A portion of the *strace* output as well as the output of the *ls* command on a test directory (/sys\_call) is displayed in figure 15.



```
root@localhost:/results
File Edit View Terminal Go Help
stat64("/sys_call", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
open("/sys_call", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
brk(0x805e000) = 0x805e000
getdents64(0x3, 0x805cfd8, 0x1000, 0x805cfa8) = 104
getdents64(0x3, 0x805cfd8, 0x1000, 0x805cfa8) = 0
close(3) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40013000
write(1, "test test.zk\n", 14test test.zk
) = 14
munmap(0x40013000, 4096) = 0
_exit(0) = ?
[root@localhost results]# ls -l /sys_call/
total 0
-rw-r--r-- 1 root root 0 Sep 2 18:13 test
-rw-r--r-- 1 root root 0 Sep 2 16:19 test.zk
[root@localhost results]#
```

Figure 15: system call trace of *ls* command on clean system

The target system was then infected with the SuckIT rootkit. The character sequence “.zk” was designated as the file hiding string as per the SuckIT README file [16]. A portion of the strace output as well as the output of the *ls* command on the test directory is displayed in figure 16.



```
root@localhost:results
File Edit View Terminal Go Help
stat64("/sys_call", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
open("/sys_call", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
brk(0x805e000) = 0x805e000
getdents64(0x3, 0x805cfd8, 0x1000, 0x805cfa8) = 72
getdents64(0x3, 0x805cfd8, 0x1000, 0x805cfa8) = 0
close(3) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40013000
write(1, "test\n", 5test
) = 5
munmap(0x40013000, 4096) = 0
_exit(0) = ?
[root@localhost results]# ls -l /sys_call/
total 0
-rw-r--r--  1 root   root           0 Sep  2 18:13 test
[root@localhost results]#
```

Figure 16: system call trace of *ls* command on a system with an installed kernel rootkit

We have demonstrated by counter example that a system that is infected with a kernel level rootkit can produce a stable definition of system calls for a particular system utility (*ls* in this case). We believe this is because information is being manipulated within the kernel as opposed to userspace. As previously mentioned, modifications to the kernel are very difficult to detect using conventional methods. As a result, we felt that dynamic program analysis was not a viable research path since it may fail to detect kernel level rootkits.

Research is currently ongoing in the area of kernel execution path analysis. This research speculates that modified kernel functionality will result in a difference in executed instructions between the modified and original kernel. Modified kernel code would most likely perform additional actions to hide secret filenames and processes from the user. These additional actions would result in additional instructions being executed within the kernel. A baseline could be built to measure the difference between the number of kernel instructions executed by a clean versus an infected system [30, 31]. This research would no longer be treating the underlying kernel operating system as a ‘black box’ but would instead be analyzing what was occurring within the black box. We believe that in the future this methodology may have merit concerning the detection of rootkit exploits.

### **3.4 Georgia Tech Methodology for Detecting Rootkits**

One methodology for detecting rootkit exploits is that which is conducted at Georgia Tech. Brian Culver, of the Georgia Tech Office of Information Technology, described this methodology to this researcher.

The Georgia Tech Office of Information Technology has the primary mission of providing technology leadership and support to Georgia Tech students, educators, researchers, administrators, and staff. OIT consists of seven directorates, including the Information Security Directorate [17].

The Information Security Directorate is responsible for numerous tasks including: educating the campus community about security-related issues, assessing current policies and developing new policies, assisting in strengthening technical measures to protect campus resources, and developing mechanisms to react to incidents and events that

endanger the Institute's information assets [18]. There are 69 separate departments at Georgia Tech with between 30,000-35,000 networked computers installed on campus. The campus has two OC-12s and one OC-48 connection to the Internet, with an average throughput of 600Mbps. Georgia Tech processes over four terabytes of data on a daily basis.

Because of the high data throughput as well as the requirement for academic freedom and the research requirements of the various departments, the Information Security Directorate does not run a firewall at the Internet connection to the campus. However, individual departments and campus agencies do run firewalls designed to meet their security requirements.

The Information Security Directorate does, at present run an Intrusion Detection System (IDS) at the campus gateway to monitor possible exploits against campus computer systems. This monitoring is done out of band and suspicious traffic is not terminated when detected. Suspicious activity will undergo a follow-on investigation.

Prior to installing the IDS the Information Security Directorate would investigate an average of five possible compromises a week. The Information Security Directorate normally received reports of these compromises from concerned computer users. Since installing the IDS, the Information Security Directorate investigates an average of five compromises a day.

When a system that may have been compromised is identified, the administrator responsible for that computer is notified that the computer system is to be investigated. This is to prevent the computer from being tampered with, which could impede the investigation.



The system is booted with a known good media disk and the hard drive is mounted in a read-only manner. The use of a known good media disk is also recommended by Chris Kuethe in his paper on detecting hacked systems [19]. A duplicate copy of the hard disk may be produced with a signature checksum in the event that the hard disk ends up being used in a criminal investigation. The techniques used by the Security Directorate personnel at Georgia Tech may be unique to Georgia Tech. The current state of the art in Computer Forensics Analysis does not provide a formal methodology for investigation [20].

The investigation begins by examining various directories a hacker may have manipulated to hide exploits on the computer system in question. The log files are examined to see if there are any records of what was done on the system. It is not uncommon for the log files to have no record of system modifications or to be deleted from the system. If the log files are deleted, steps are taken to try to retrieve them. Next, previously known directories where a hacker may choose to hide exploit files are examined. The chkrootkit tool may be run to check if a rootkit has been installed on the system. If these checks prove unsuccessful, the Security Directorate personnel then conduct a more detailed examination of the system. For example, on a UNIX or LINUX operating system, commands such as *'find'* or *'locate'* are used to try to find directories that may have been used by the hacker when the system was exploited.

If such a directory is located, then a listing of that directory occurs to see what files are present in that directory. The *file* and *strings* command are used on these files to examine them. The *file* command is run to try to determine the file type. The output of the *strings* command is read to try and recognize any suspicious text strings that may

indicate what exploit was done to the computer.

The `/proc/` directory is then checked to see if a program is running in memory. The process id numbers (pids) are compared between those listed by the report process status (*ps*) command (using the `-ef` switch) and those listed in the `/proc/` directory. A difference between these two listings indicates that the *ps* command was most likely modified by the hacker to hide the processes that the hacker has running on the computer. The Information Security Directorate personnel use the `/proc/` directory as a true listing of what is currently running on the system being investigated. The processes that show up in the `/proc/` directory but that are not listed by the *ps* command are examined using the *file* and *strings* command.

The *strace* command may be used to trace system calls for suspicious program binaries left on the system by the hacker. The print shared library dependencies (*ldd*) command may also be used to check on shared library dependencies of the suspicious programs, especially for those suspect programs that have the same name as known good system binaries. A difference in library listing is determined to be a direct indication that a hacked version of system binaries is installed on the system.

A similar methodology is used for other operating systems to determine if the system has been exploited by a rootkit. We believe it is the case that in general, information security personnel have no formal methodology to determine if a computer has been infected with a new unknown or previously modified known rootkit without conducting an exhaustive manual investigation of the exploit [21].

### **3.5 Summary**

We have provided a current overview of the current state of rootkit detection methodology. We described and analyzed various GPL tools that are in use today by system administrators. Our analysis identified the strengths, weaknesses and shortcomings of these tools. The feasibility of conducting Black Box analysis to identify possible rootkit exploits was then examined. A shortcoming of this method was demonstrated concerning the detection of kernel rootkits. The current detection methodology that is in use at Georgia Tech was then examined. The current state of the art in detecting rootkit exploits consists of a mixture of automated procedures as well as the manual examination of systems suspected to be infected with a rootkit exploit. Our research indicates that there is no formalized method currently in existence to detect or characterize rootkit exploits. It is a goal of this research to develop such a method.

## Chapter 4

### **Formal Methodology to Characterize Rootkits**

There is no current methodology at present to characterize rootkit exploits as existing, modifications to existing, or entirely new. We believe that such a methodology would assist network system administrators in increasing the overall security level of their networks. Requests have been made for such a methodology [32]. Our initial interest in this research area came about as a result of a discussion with the Technical Project Director for Security at the Georgia Institute of Technology. System administrators would be able to characterize the threats that their networks would face. For example, if a system administrator knew that a certain type of already known rootkit posed a threat to the network, that system administrator could take the appropriate measures that had been used previously to protect the network. Appropriate measures could also be taken for modifications to existing rootkits. Previous defensive measures could be utilized to protect a network if these measures could be validated against the modified rootkits. New methods could be developed for new capabilities that were introduced into these modified rootkits. New defensive measures could also be developed for rootkits that are entirely new. A system administrator could possibly associate a particular type of rootkit with particular exploit and then defend the network against this type of rootkit. New signatures could also be developed as a result of classifying rootkits as either modification to existing or entirely new. Signature based intrusion detection systems must know a particular signature to classify an intrusion. Such a signature may not exist for modifications to existing or entirely new rootkits. Programs such as chkrootkit are for

the most part signature dependent when they attempt to detect rootkit exploitation. A formal methodology that is capable of classifying rootkit exploits as existing, modification to existing, or entirely new is capable of providing new signatures for programs such as chkrootkit. The classification of rootkits in this manner may also result in the identification of specific anomaly patterns that can be associated with a particular rootkit.

The need for a formal methodology to characterize software weapons has already been addressed in previous research [33]. This research attempts to develop a taxonomy, or formal classification method for software weapons. A premise of this research is that a research area will benefit from a structured categorization process. We agree with this and the goal of our research is to develop a detection and classification methodology for a specific class of software weapon known as a rootkit which we have previously defined. This rootkit detection and classification methodology will result in techniques to characterize rootkits as well as new methods to detect them. This methodology can be used throughout both the research and operational communities to increase the overall level of security of the Internet.

We have developed a mathematical framework to define rootkit exploits as either existing, modification to existing, or entirely new. This is necessary so that we have some formal method to classify rootkits. Rootkits are a unique subclass of software exploits and Trojan Horse type programs. A true rootkit should maintain the original functionality of the computer program or capability that it is intended to replace. We use this characteristic in our classification and detection methodology. We then describe the methodology in detail to include the establishment of a testing platform and the steps that

are necessary to perform our analysis to identify characteristics that can be used for rootkit detection and classification.

#### **4.1 Mathematical Framework to define Rootkits**

Methods have already been developed to attempt to classify various types of computer exploit code. In Thimbleby, Anderson and Cairns [1] a preliminary framework for modeling Trojans and computer virus infections was developed. This work dealt with the general case of viruses and Trojans. This dissertation makes use of the ideas presented in that work to develop a mathematical framework to classify rootkit exploits. The focus of this work is more specific in that it develops a method to classify rootkits as existing, modification to existing, or entirely new.

A computer virus has been defined as a computer program that is able to replicate all or part of itself and attach this replication to another program [22]. The type of rootkits that we wish to classify does not normally have this capability so this is not a method that we could use to detect or classify rootkits. A true rootkit program that is intended to replace an existing program or capability on the target system must have the same functionality as the original program or capability plus some increased functionality that has been inserted by the rootkit developer to allow backdoor root-level access and/or the ability to hide specified files, processes, and network connections on to the target system. This increased functionality is provided by added elements contained within the rootkit. The increased functionality of the rootkit, with its associated elements, provides a method that can be utilized to detect and classify rootkit exploits. Rootkits can be characterized by using a variety of methods to compare the original program to the rootkit program and identify the difference, or delta ( $\nabla$ ) in functionality between the two programs. This

$\nabla$  can serve as a potential signature for identifying the rootkit.

It has been recognized that evaluating a program file by its CRC checksum is both faster and requires less memory than comparing a file by its contents [8]. The results of this comparison will only tell you that a current program file differs from its original program file. Using this check to detect rootkits would not tell you if this rootkit is an existing, modification to existing, or entirely new rootkit exploit. It is also recognized that Trojan Horse type programs can be detected by comparing them to the original program file that they are intended to replace [8]. The approach we choose to follow is that rootkits can be classified comparing their  $\nabla$  against previously identified  $\nabla$ 's of known rootkits.

For our framework we assume that we have already identified a program as being part of a potential rootkit. We are able to do this by using a Honeynet to capture rootkits. In addition, we have a copy of the original programs that the rootkit replaced. From our definition of a true rootkit we can assume that these two programs are indistinguishable in execution since they will produce similar results for most inputs. Therefore, these two programs are similar to each other. From [1], we recognize that similarity is not equality, we may not be able to recognize that the programs differ in the amount of time that we have available to analyze them. Two programs are indistinguishable when they reproduce similar results for most inputs. A true rootkit should therefore be indistinguishable from what it is intended to replace since it should have the same functionality as the original programs it is to replace in addition to the new capabilities that were added by the rootkit developer.

We also use the quantifiers, *similarity* ( $\sim$ ), indistinguishable ( $\approx$ ), and the meaning of a

program  $[[\bullet]]$  that was presented in [1] and define them in a similar manner.

- $\sim$  (similarity) – a poly log computable relation on all possible representations (defined as  $R$ ) of a computer to include the full state of the machine consisting of memory, screens, registers, inputs, etc. A single representation of  $R$  is defined as  $r$ . Poly log computable is defined as a function that can be computed in less than linear time meaning a representation can be evaluated without having to examine the entire computer representation.
- $\approx$  (indistinguishable) – two programs that produce similar results for most inputs.
- $[[\bullet]]$  (the meaning of a program) – what a program does when it is run

We presume to have two programs:  $p1$ , the original program, and  $p2$ , identified as malicious version of program  $p1$  that provides rootkit capabilities on the target system. If  $p2$  is part of a true rootkit then  $p1$  and  $p2$  are indistinguishable from each other. These two programs will produce similar outputs for most inputs. In a manner similar to [1] we can state that  $p1$  is indistinguishable from  $p2$  if and only if

$$\text{for most } r \in R : [[p1]]r \sim [[p2]]r \Rightarrow p1 \approx p2$$

meaning for most representations of a machine out of all possible representations the results of program  $p1$  are similar to the results of program  $p2$  which implies that  $p1$  is indistinguishable from  $p2$ .

We will now apply set theory to show a method to characterize rootkit exploits. We assume to have the following programs:

$p1$  – original set of programs

$p2$  – malicious version of programs that replace  $p1$  programs

If  $p2$  is a true rootkit of  $p1$  then we can state that  $p1$  is a subset of  $p2$  since all of the



elements that exist in  $p1$  must exist in  $p2$ . Then  $p1$  is a proper subset of since all elements of  $p1$  exist in  $p2$  but  $p1$  is not equal to  $p2$ , This can be written as:

$p1 \subset p2$ , since  $p1 \subseteq p2$  and  $p1 \neq p2$  meaning  $p2$  has at least one element that does not belong to  $p1$ .

We will now identify the difference between  $p1$  and  $p2$ .

$p2 \setminus p1 = p'$  is the difference between  $p2$  and  $p1$  containing only those elements belonging to  $p2$ . This is the  $\nabla$  that we have previously discussed.

We assume we have identified another rootkit of  $p1$  and call this  $p3$ . We can identify this collection of programs as a rootkit of type  $p2$  as follows:

If  $p3 - (p' \cap p3) = p1$  then  $p3$  contains the same elements as program  $p2$  and is the same rootkit.

If the preceding statement is not true but elements of  $p'$  are contained in  $p3$ , written as  $p' \in p3$ , then we can assume that  $p3$  may be a modification of rootkit  $p2$ . If there are no elements of  $p'$  in  $p3$ , written as  $p' \notin p3$ , then we may assume that  $p3$  is an entirely new rootkit.

A rootkit may not be a true rootkit in that it does not exact incorporate the capabilities of what it is intended to replace. However, we can still use our methodology to classify rootkits. We can identify the differences ( $\nabla$ ) between the rootkit and whatever the rootkit is intended to replace and use this  $\nabla$  in our analysis. This  $\nabla$  can be used to both detect and classify rootkit exploits.

We will follow this framework to classify the rootkits that we are examining. We are examining numerous rootkits as a part of our research.

## 4.2 Methodology to Characterize Rootkit Exploits

To be able to characterize rootkits with our methodology we must have a valid copy of the rootkit. We will subsequently address the method (Honeynet) that we propose using to acquire rootkits as well as a keystroke capture of the hacker's activity on the target system. This information is available to us for analysis but at this point we assume that we have a copy of the rootkit that the hacker intended to install on the target system as well as a record of the hacker's activity. We also know the specific target operating system that was to be the intended target of this rootkit exploit. In some cases we will have the source code of the rootkit which will greatly simplify our analysis. In other cases we will only have the binary executable files for the rootkit exploit. The following is a description of the methodology that we follow to identify the specific  $\nabla$  of a rootkit exploit targeting the Linux operating system. We choose Linux as our operating system for this research but this methodology should apply to other UNIX type operating systems.

1. Start with a clean installation of the specific kernel version of the operating system that was the intended target of the rootkit exploit.
2. Install a kernel level debugger on this system. The installation of the kernel level debugger will probably require the system to be recompiled with a custom kernel.
3. Install and run a file integrity checker program on this system. Select target directories based on an analysis of the installation that occurred when the rootkit was originally acquired.
4. Install a rootkit detection program such as chkrootkit on the target system. This

will help us to detect many existing system utility rootkit exploits and may help us to identify modification to existing as well as entirely new rootkit exploits.

5. Install a program such as kern\_check to check the integrity of the system call table within the kernel. Run this program on the target system to establish a baseline and ensure that the kernel integrity has not been compromised on this initial installation.
6. Make a copy of the kernel text segment of memory via /dev/kmem for future comparison. The kernel text segment of kernel memory should remain consistent for a particular kernel build. Any deviation between this copy and a future copy could indicate that the kernel may have been compromised by a kernel level rootkit exploit. A more detailed analysis of kernel space can then be conducted via the kernel debugger (kdb) program that we have previously installed
7. Run the file integrity checker program and the rootkit detection program on the target system prior to infecting the system with the rootkit to establish a baseline for comparison between a clean and infected system.
8. Install the rootkit on the target system. Follow the installation steps that were used from when the rootkit was initially acquired for analysis.
9. Run the file integrity checker program on the system and note the results. The presence of certain types of rootkits should be indicated by the results of the file integrity checker program but other types of rootkits, specifically those that target the kernel, may not be detected by this type of program.
10. Run the rootkit detection program (chkrootkit) on the system that has been

infected with the rootkit that is being analyzed. If this is a previously known system utility rootkit then a program such as chkrootkit should be able to detect the presence of this rootkit. If the file integrity checker program detected a change to a system utility binary program file but the rootkit detection program did not detect the presence of a rootkit then we can assume that we are dealing with either a modification to an existing system utility rootkit or an entirely new system utility rootkit.

11. Run the kernel integrity check program on the target system. If the system call table was modified or redirected, then this program should be able to detect what has been modified. Make note of each system call that is indicated as being modified as well as a total count of the number of system calls that have been modified by this rootkit. The system calls that a rootkit modifies can establish a signature for a specific kernel level rootkit [34]. This program may not be able to detect some other modification to the kernel text code segment of the kernel.
12. Make a copy of the kernel text code and compare it against the original copy of the kernel text code that we prepared before infecting this system with the rootkit. A difference between these two files may indicate the presence of a kernel level rootkit. This is especially significant if the existing kernel integrity checker programs failed to detect any modification to the kernel. This would indicate that we are dealing with a new type of kernel level rootkit that does not target the system call table. The system call table is the normal avenue of attack for hackers who are attempting to create a kernel level rootkit.

The results of these steps can be used to classify a rootkit exploit as an existing,

modification to an existing, or an entirely new rootkit. The following figures show the steps to follow to collect this information.

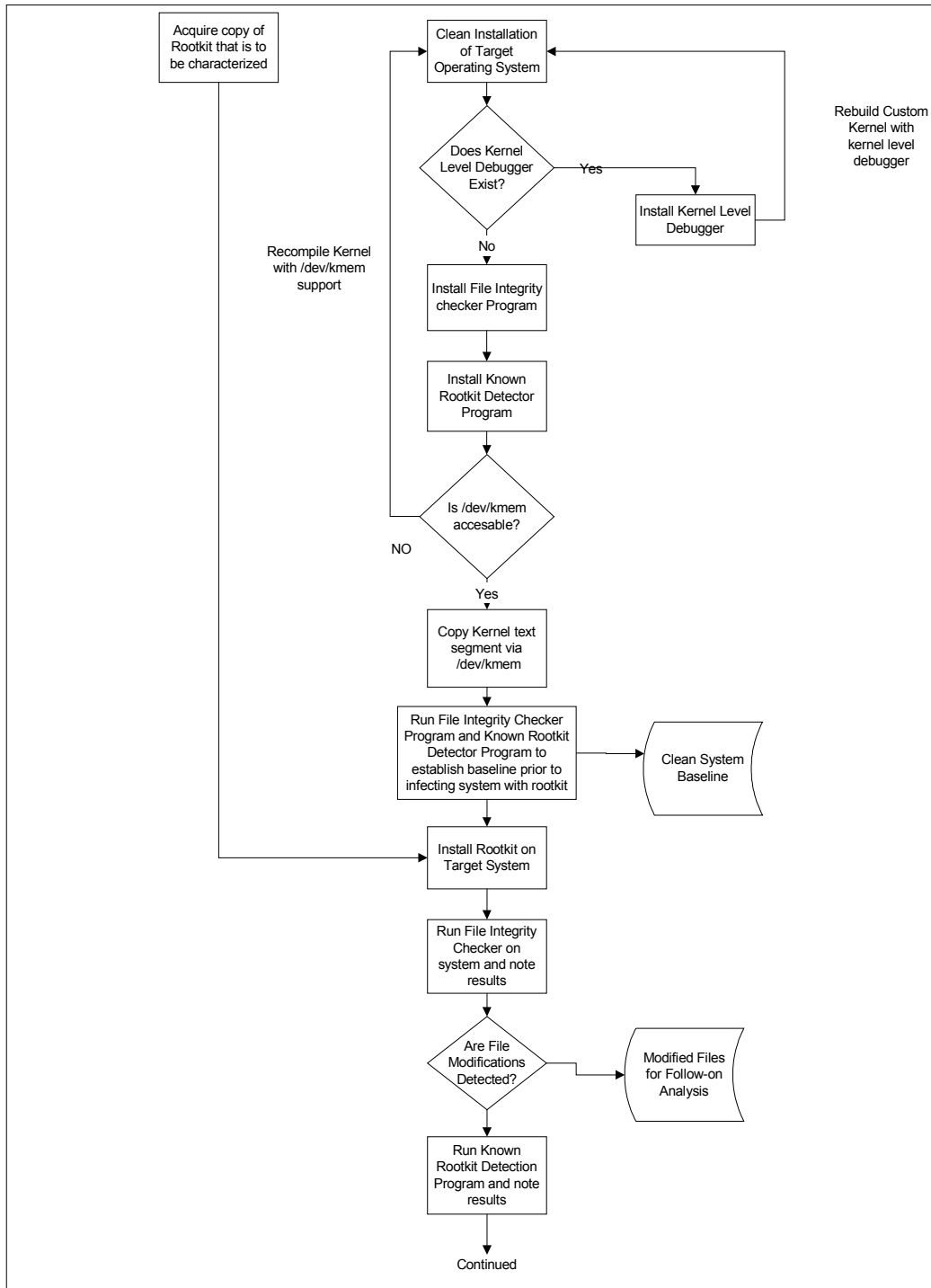


Figure 16: Flowchart of Characterization Methodology part 1

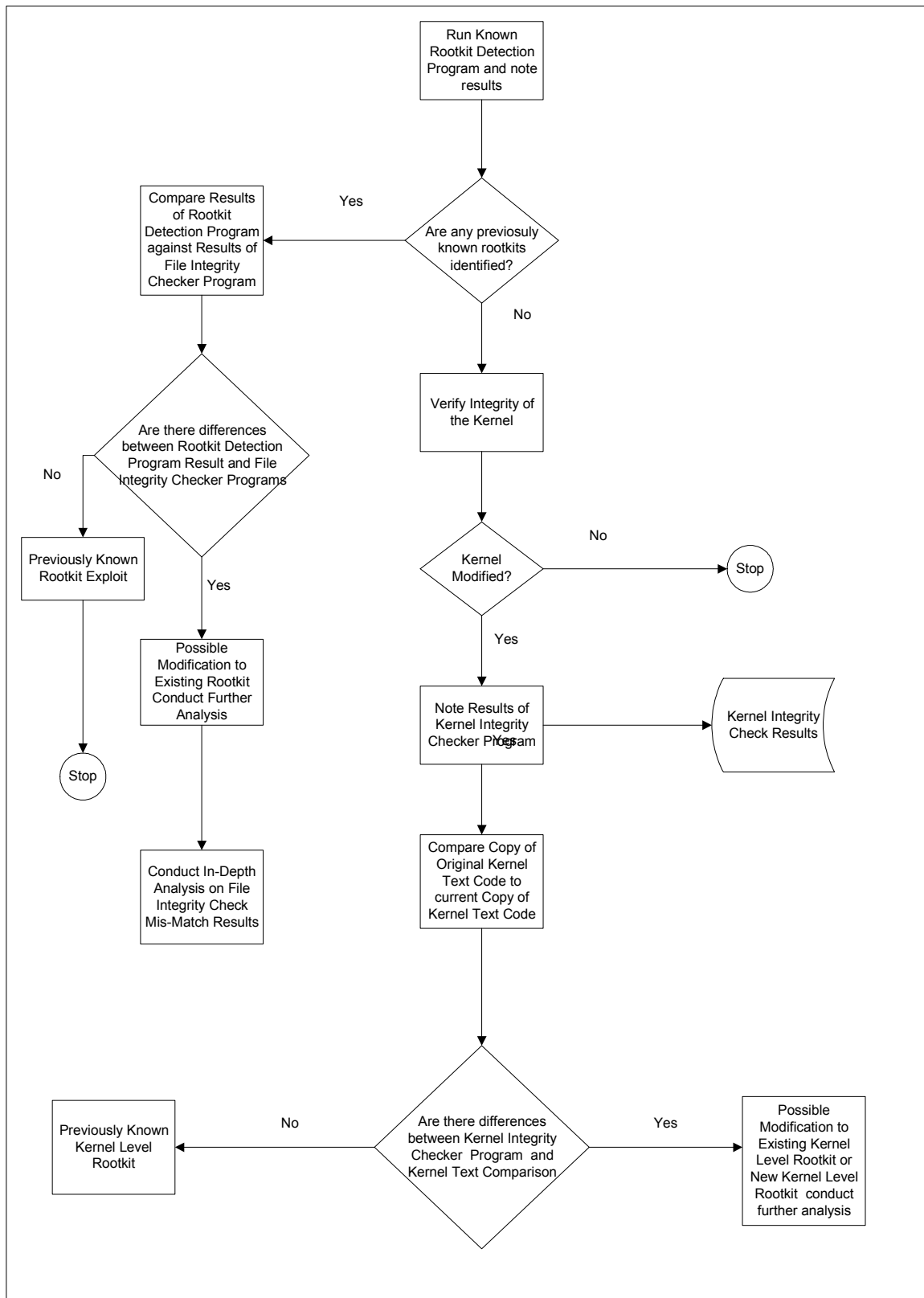


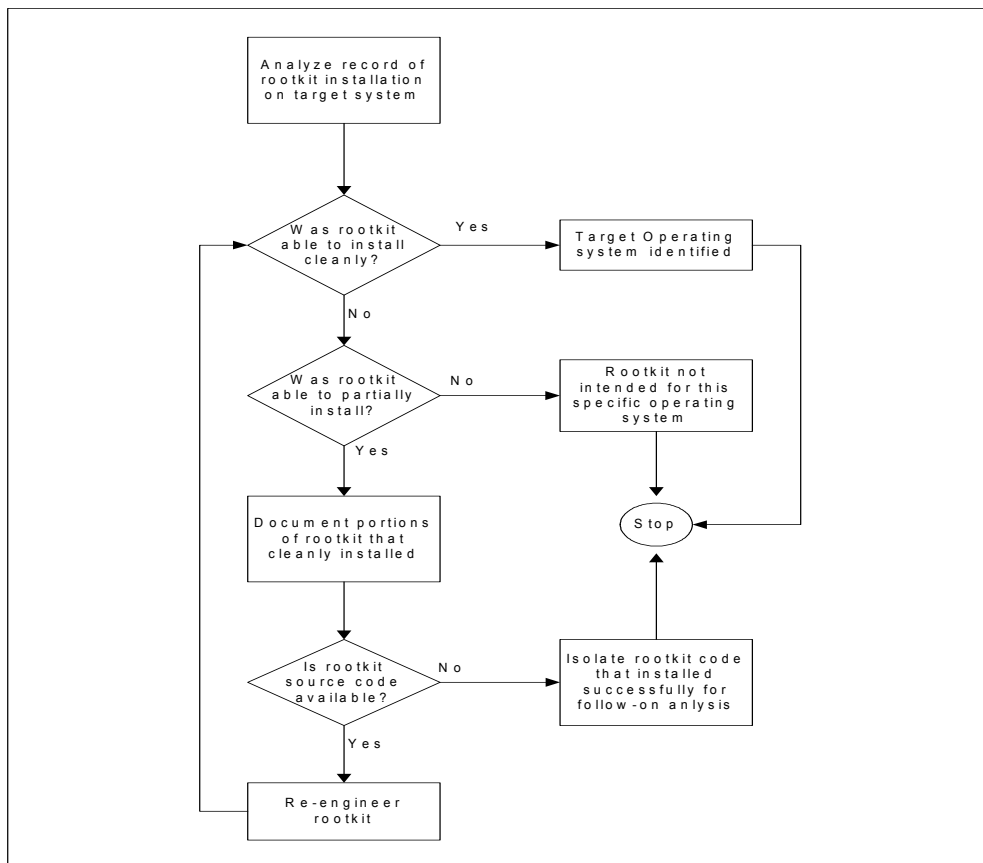
Figure 17: Flowchart of Characterization Methodology part 2

We will now describe each step of the methodology in detail.

#### **4.2.1 Clean Installation of Target Operating System**

A particular operating system must be chosen to conduct a complete analysis of a rootkit exploit to determine if it is existing, modification to existing, or entirely new. This decision can be based on a number of factors to include any available documentation that is available with the rootkit to include README documentation or any install programs (configure, Makefile). The decision may also be based on the target of the initial rootkit attack. Some rootkit exploits are successful in installing on a target operating system while others may not install successfully. If the rootkit was able to install cleanly on a target system when it was initially acquired, then that is the operating system that should be used to conduct the analysis. A clean rootkit installation is indicated by a system that functions as normal without any indication of the underlying rootkit. If the rootkit was only able to partially install or unable to install on the target system, then a decision must be made concerning further analysis. Many unsophisticated hackers (known as 'script kiddies') will blindly attempt to install a particular operating system rootkit on any system that they are able to gain root access. This same behavior can also be observed in automated Internet worm attacks. Other rootkits may have been built for a particular version of a kernel and will fail to install cleanly or not install at all on the target system. A decision must be made by the analyst to determine if the actual rootkit code is to be re-engineered to install on the target system so that the analysis results can be included in the methodology. This decision can be based on any documentation that is available with the rootkit exploit as well as whether the rootkit was able to partially install on the target system. A rootkit that was able to partially install on

the target system is a good candidate for re-engineering and further analysis. It is recommended to only attempt to re-engineer the rootkit code if the initial source code for the rootkit is available to the analyst. Our research is focused on rootkits targeting the Linux operating system. The source code for the rootkits that we examined were written in the C programming language. The ability to re-engineer rootkit source code requires an in-depth knowledge of C programming and a familiarity with the Linux operating system. This ability has proved beneficial in our analysis of rootkit exploits by providing us with background information on the skill and logic of the rootkit developer. The following figure demonstrates in detail the process of selecting an operating system for analysis.



**Figure 18: Clean Installation of Target Operating System**



If the rootkit could not be installed on the particular system that it was originally targeted for, the analyst may choose to attempt to install the rootkit against some other operating system. The decision can be made concerning the amount of documentation that is available with the rootkit. A rootkit such as this can still be archived for future reference.

#### **4.2.2 Use of a Kernel Level Debugger**

If a kernel level debugger exists for the target operating system that has been identified in the previous step, then it should be installed. A kernel level debugger will assist in analysis of the underlying kernel for kernel level exploits. Certain portions of the kernel should be immutable for various installs of a particular operating system. Differences within these portions of code may indicate that the kernel has been compromised. The man page for kdb describes its capabilities as follows: “This debugger allows the programmer to interactively examine kernel memory, disassemble kernel functions, set breakpoints in the kernel code, and display and modify register contents.” [35]. The features that we are interested in for conducting analysis are the *md* command, which allows for the analysis of kernel memory, and the *id* command, which allows for the disassembly of kernel functions.

If a kernel level debugger is not available for the target system than it still may be possible to debug the kernel using the normal gdb debugger with kgdb. Using kgdb requires the use of two computers to analyze the kernel while the use of kdb only requires one computer. kgdb is described as a source level debugger while kdb is described as an assembly language debugger [36]. Versions of kgdb exist for various UNIX type operating systems. We did not investigate the use of kgdb in our research.

### **4.2.3 Install File Integrity checker Program**

We have already discussed the GPL file integrity checker program AIDE in a previous section of this thesis. We will use this file integrity checker program to establish a baseline, or ‘snapshot’, of the system before the system is infected with the rootkit exploit[9].

### **4.2.4 Install Known Rootkit Detector Program**

The GPL chkrootkit program is also a program that we have previously described. This program is useful for detecting rootkits that are already known. We will use this program to identify existing rootkits as well as potential modifications to existing rootkits.

### **4.2.5 Access to /dev/kmem**

The /dev/kmem file is a special file known as a device file that exists in most UNIX and UNIX-like operating systems to include Linux. This device file allows a user with root privilege access to the kernel memory [37]. It is this capability that makes the /dev/kmem file a target for hackers that gain root access on a compromised system. A hacker with root permission has the capability to modify the underlying kernel code.

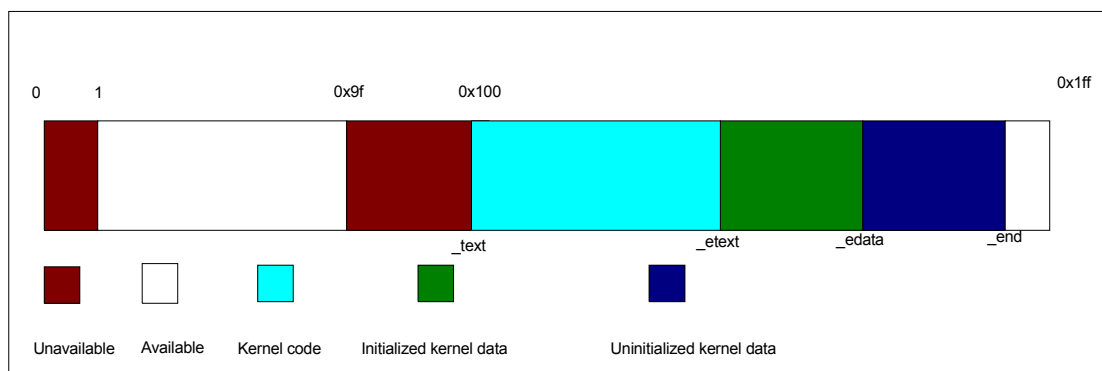
The /dev/kmem device file exists to provide access to the virtual memory address space for the current process as it is seen by the underlying kernel. Virtual addresses within the kernel can be targeted for reading and/or writing with the *lseek* command. The /dev/kmem device file on Linux supports the following commands: open, close, read, and write [38].

If the /dev/kmem file is not currently available for the target operating system then this system should be recompiled with /dev/kmem support. Some system administrators

choose to set `/dev/kmem` to read-only to secure their system. However, there is a workaround to this that would allow a hacker to modify the kernel via `/dev/kmem` even if write access is not available for `/dev/kmem` [39].

#### 4.2.6 Copy kernel text segment via `/dev/kmem`

The immutable kernel code, known as kernel text, is loaded into kernel memory and is accessible via `/dev/kmem`. As previously explained, this code should remain consistent across the same kernel version of a particular operating system unless that version of the operating system is patched and recompiled. The following figure shows how this kernel code is stored in kernel memory.



**Figure 19: first 512 page frames of first two megabytes of kernel memory [40]**

We are concerned with the section of code between the `_text` and `_etext` identifiers in the above figure. The first byte of kernel code is designated by the `_text` symbol and the end of the kernel code is indicated by the `_etext` symbol. Initialized and uninitialized kernel data follows in the remaining two segments. The symbols in the above figure are produced upon compilation of the kernel and written to the `System.map` file. We can retrieve the addresses of the `_text` and `_etext` symbol from this file. We can then use

these addresses to make a copy of the original kernel text region of memory to establish a baseline for future analysis. This technique is similar to that of the GPL StMichael program that we have previously discussed in Section 3 of this paper. As a loadable kernel module (lkm), StMichael makes a copy of this segment of kernel code within the kernel and conducts a periodic examination of this archived copy of the kernel code against the currently installed kernel code. However, StMichael modifies this section of kernel code and is in essence a kernel level rootkit. Our method makes a copy of the kernel code we wish to use as a baseline and stores this data within userspace. Once a copy of this code is produced, it can be copied to read-only media for follow-on analysis.

#### **4.2.7 Establish Baseline on system prior to infection with rootkit**

This is the point where a clean baseline for the target system can be established prior to infecting the system with a rootkit. All of the necessary tools have been installed on the system and the results of a baseline establishment should be that of a clean, uninfected system. The baseline of the file integrity checker program is used subsequently to detect evidence of system utility rootkit infection on the target system. The resulting database that is produced by the file integrity checker program should be archived to a read-only media for follow-on analysis once the system is subsequently infected by a rootkit. In addition, the binary system utilities that are used by the file integrity checker program should also be copied to the read-only media for use during the follow-on analysis. This procedure is recommended in general by the developers of the GPL chkrootkit known rootkit detection program [3].

The selection of directories to use in establishing a baseline on the target system should be based on any documentation that is available from the rootkit. This documentation

can be used to determine what system utilities the rootkit may target for replacements. As a default, the /bin, /usr/bin, and /sbin directories should be selected since many potential system utilities that are targeted by hackers reside in these directories.

The known rootkit detection program should also be run against the target system at this point. This check should not show any indication of rootkit exploit of the target system since we are starting with a clean system that has not yet been infected with a rootkit. If the presence of a rootkit is detected, both the target system and the rootkit detection program should be investigated prior to proceeding further in analysis. This could be an indication of a faulty install of the operating system or a mismatch between the operating system and the chkrootkit detection program. Figure 20 demonstrates the steps necessary for establishing a baseline for the target system prior to rootkit infection.

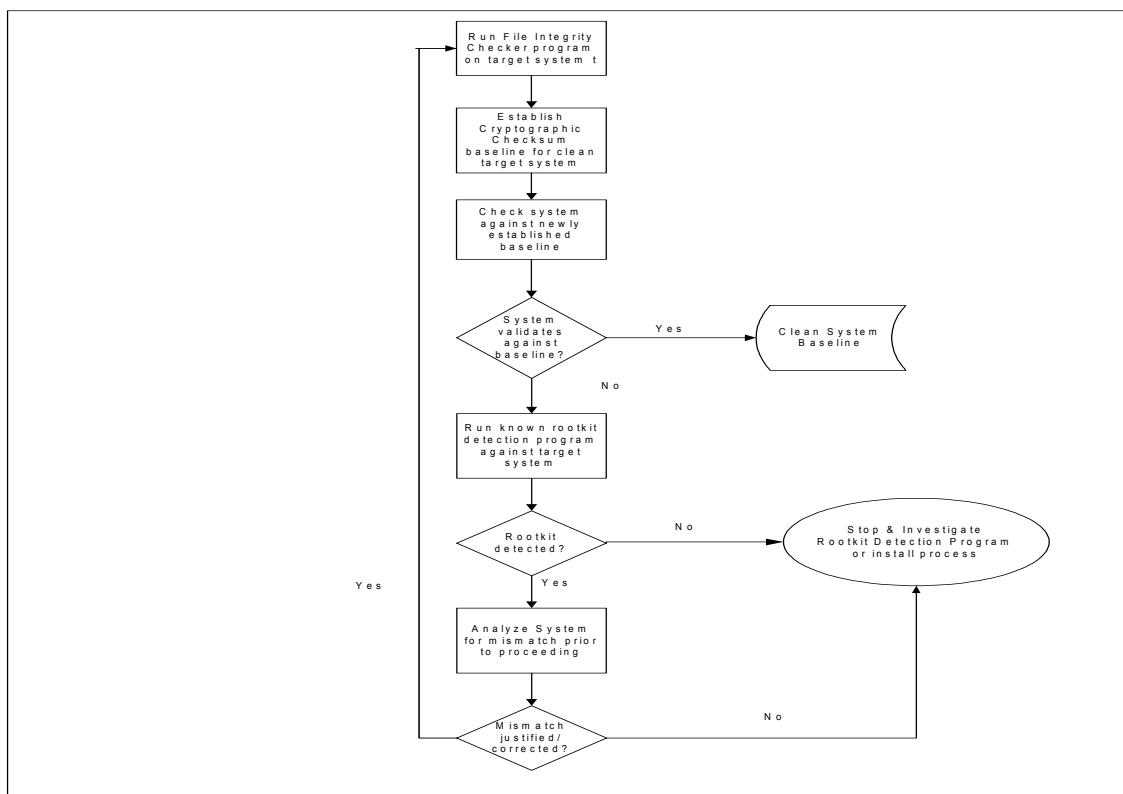


Figure 20: Baseline Establishment

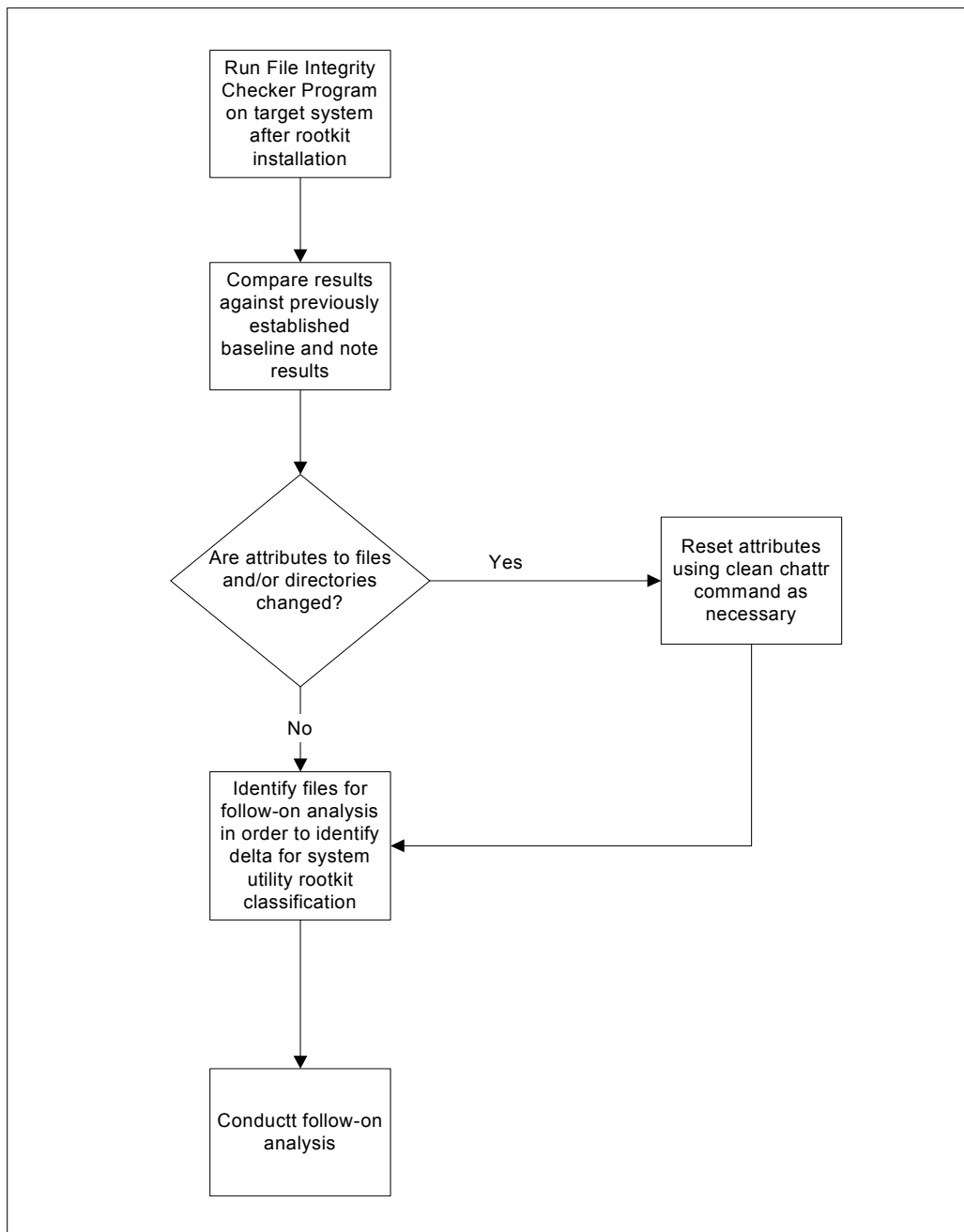
#### **4.2.8 Install rootkit on target system**

At this point a rootkit that is able to cleanly (or partially) install on the target system is available. The installation should occur as per any install documentation that was provided with the rootkit (configure and Makefile) or as per any other install script that was used by the hacker when the initial system was compromised. An original record of the system compromise was already collected from the Honeynet, even if the hacker deleted any trace of the rootkit installation on the target system.

#### **4.2.9 Run file integrity checker**

The file integrity checker should now be run on the target system after the installation of the rootkit. Any changes that are detected are to be documented for further analysis. It is significant to note that certain rootkits may only change attributes to certain files and directories without changing the underlying files and directories. This will result in the file integrity checker program indicating that these files and directories have been changed. Our analysis indicates that a rootkit developer may choose to change the underlying attributes to a file or a directory to prevent the changes that they have made to the target system from being undone. It is possible to set the attributes of a file so that these files can not be subsequently modified or deleted (making these file unchangeable (-i bit) or undeletable (-u bit)), even by a user with root or administrator privilege. Figure 8 of section 3.1 displays an example of this. This can be accomplished with the change attributes on file system (*chattr*) system utility command. A system administrator can reset the permission on any files modified by the rootkit with the *chattr* command. However, the hacker may have modified this command with the rootkit so that it will no longer function correctly on the files that have been modified. The *chattr* and list

attributes on file system (*lsattr*) are two system utilities that should be archived from the clean system installation for future use as discussed in section 4.2.7. The following figures demonstrates the process of running the file integrity checker on the target system after it has been infected with the rootkit exploit.



**Figure 21: File Integrity Check Procedures**

Any files that are identified as having been modified are candidates for follow-on analysis to identify a  $\nabla$  (delta) for reference. This  $\nabla$  can be compared against previous  $\nabla$ 's from already known rootkits to determine if this rootkit is existing, modification to existing, or entirely new. We will present our classification methods in a subsequent section of this document.

Files and directories that had their attributes changed by the *chattr* command but which were otherwise unmodified can have their attributes reset using a clean copy of the *chattr* system utility command. Documentation of hacker install scripts can be analyzed for use of this command in the installation of the rootkit or these files can be analyzed prior to rootkit installation to determine the original attributes to these files. This may be necessary to allow for follow-on analysis on the target system.

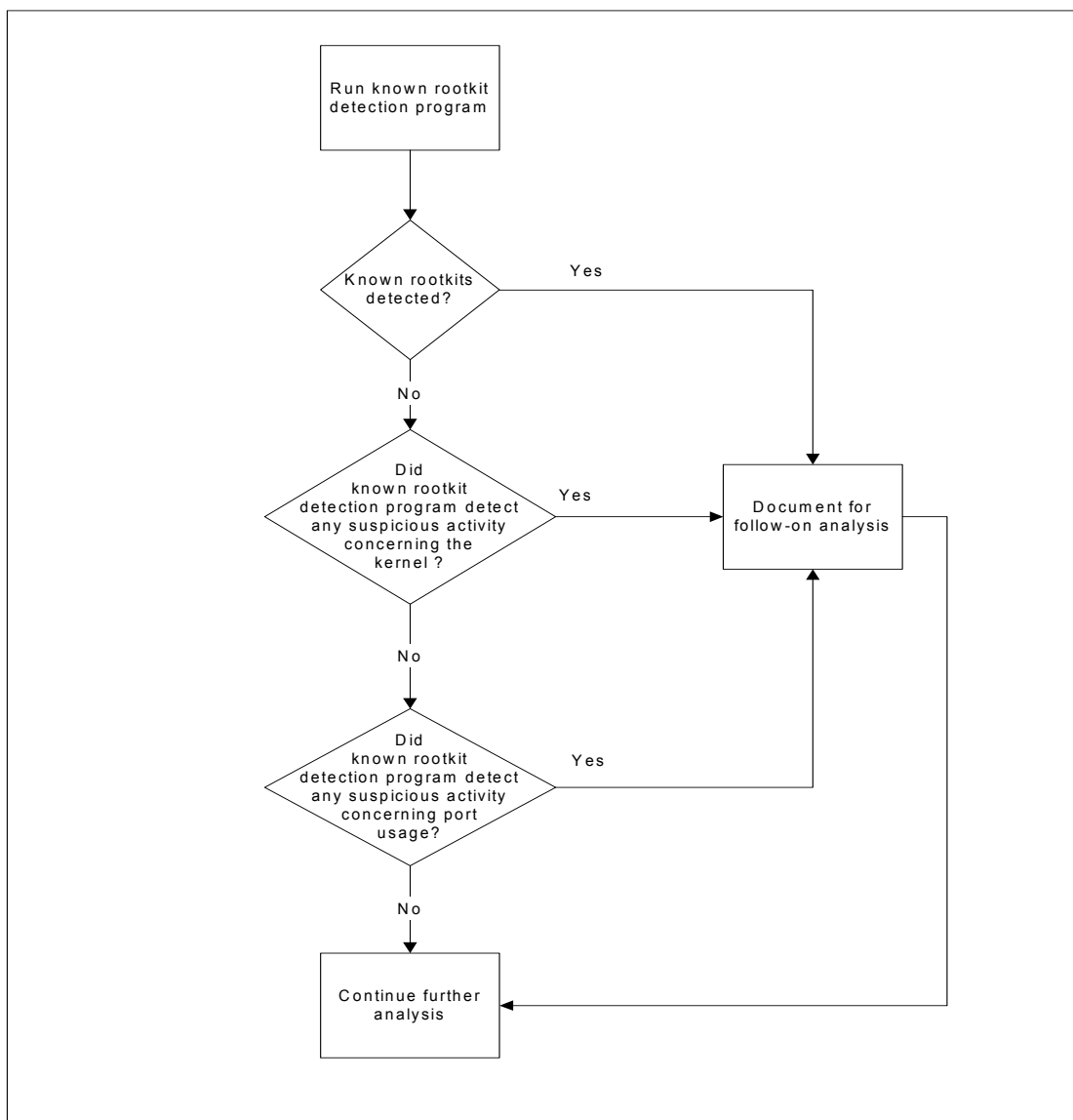
#### **4.2.10 Run Known Rootkit Detection Program**

The known rootkit detection program 'chkrootkit' should now be run on the target system. Most previously known rootkits, as well as modifications to previously known rootkits that maintain many of the same characteristics as the original rootkit exploit should be detected by this program. The chkrootkit program may indicate that the system is infected by a particular rootkit when in fact the system is infected by some entirely new rootkit that shares the same characteristic signature as the original rootkit. This signature has been previously identified and incorporated into the chkrootkit program.

The chkrootkit program may also indicate some suspicious activity on the target system that it is unable to classify as a particular rootkit exploit. This may be the case concerning kernel level rootkits. Figure 9 of section 3.1 is an example of this. A



mismatch was detected between the /proc directory and the report process status (*ps*) command by the chkrootkit program. This is an indication the kernel has most likely been compromised. The chkrootkit program will also detect that certain ports on the system are associated with possible trojan programs. The following figures demonstrates the process of conducting known rootkit detection analysis on the target system after it has been infected with the rootkit exploit.



**Figure 22: Known rootkit detection analysis**

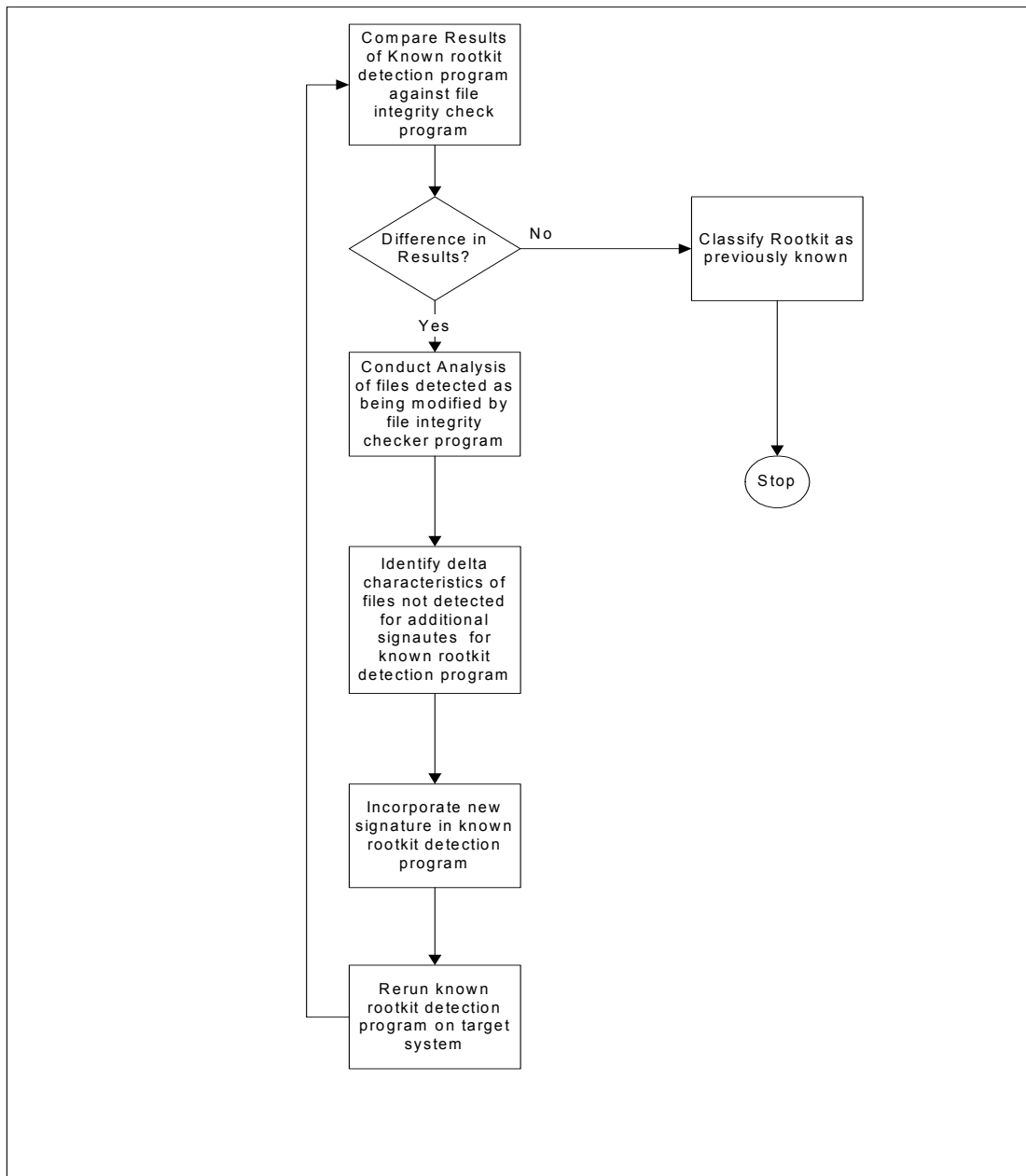
The results of this step in the analysis process are used in follow-on analysis to determine if the rootkit being examined is existing or modification to existing. Entirely new rootkits will normally not be detected by this step in the analysis unless they share attributes that are similar to existing rootkits. Suspicious kernel activity that is detected by the chkrootkit program is noted for further analysis at the point where the kernel integrity is checked.

#### **4.2.11 Previously known rootkit identification**

In our analysis we conducted a comparison using the results of the previous step (run known rootkit detection program) against the results of the file integrity checker program. The files that a previously known rootkit modifies should already be published. If this is the case and we are dealing with a previously known rootkit, the files that are detected as modified by the file integrity checker program should be the files that are detected as being modified by the known rootkit detection program. Any deviation from this may indicate that we are dealing with a modification to an existing rootkit.

A modification to an existing rootkit may change additional files besides those that are changed by the original rootkit. This is indicated by files that are detected as being changed by the file integrity checker program but not detected by the known rootkit detection program. A modification to an existing rootkit may change the same files as the original rootkit in a new manner, resulting in these modified files no longer being detected by the known rootkit detection program. At this point an in-depth analysis of the files indicated as being changed by the file integrity checker program but not detected by the known rootkit detection program should occur to identify  $\nabla$  characteristics for incorporation in subsequent versions of the known rootkit detection algorithm. It is

significant to note that rootkit modifications of the kernel will most likely not be detected by this analysis. Kernel analysis for rootkit exploitation is addressed in a subsequent section. The following figures demonstrates the comparison that must occur to determine if we are dealing with an existing rootkit or a modification to an existing rootkit.



**Figure 23: Previously known rootkit identification**

#### **4.2.12 Verify Integrity of the Kernel**

The system should now be analyzed to detect if the rootkit has made any modifications to the kernel. As previously discussed in section 2.4, current kernel rootkits target the System Calls. Unix type systems implement most interfaces between user mode processes and the underlying hardware devices that the user mode process needs to access with system calls. The system calls are issued to the kernel by the user mode process [41]. The ability of compromised system calls to provide erroneous information to user mode processes makes system calls a target of rootkit developers. As previously mentioned, kernel rootkits are difficult to detect using conventional detections methods.

Our methodology initially focuses on analysis of the system calls to determine if the kernel has been modified by a rootkit. The first step is to verify the integrity of the System Call table within the kernel. The system call table within the kernel must be checked for modifications to system call addresses as well as being checked to see if a redirection of the system call table is occurring within kernel memory. Redirection of the system call table results in a new system call table being created in user space with a corresponding new address. All subsequent references to the system call table are directed to this new corrupted system call table while the original system call table remains intact. Analysis of the system call table to detect modification to system call addresses can be conducted by comparing the addresses of the system calls listed in the `/boot/System.map` file with the addresses of the system calls in the system call table (`sys_call_table`). This analysis can be done using the `kdb` kernel debugger program as well as with the GPL tool `kern_check` described in section 3.1. This type of compromise can be easily detected if the proper areas of kernel memory are examined. The area of

kernel memory to check for this type of exploit is the `sys_call_table`. The address of the `sys_call_table` within kernel space is available within the `System.map` file once the system is compiled. This address will remain consistent for all of the same versions of kernel code.

To detect the presence of a rootkit that redirects the `sys_call_table` other areas of the kernel space must be examined. A kernel level rootkit may overwrite the location in kernel memory that contains the address of the system call table. The kernel level rootkit is able to accomplish this by querying a specific register within the processor. It then uses this information to find the entry point address within the kernel for the system call table and overwrites this address with the address of a new system call table containing the addresses of some malicious system calls that the rootkit also creates. We present an in depth analysis of how a particular rootkit accomplishes this within an appendix of this thesis.

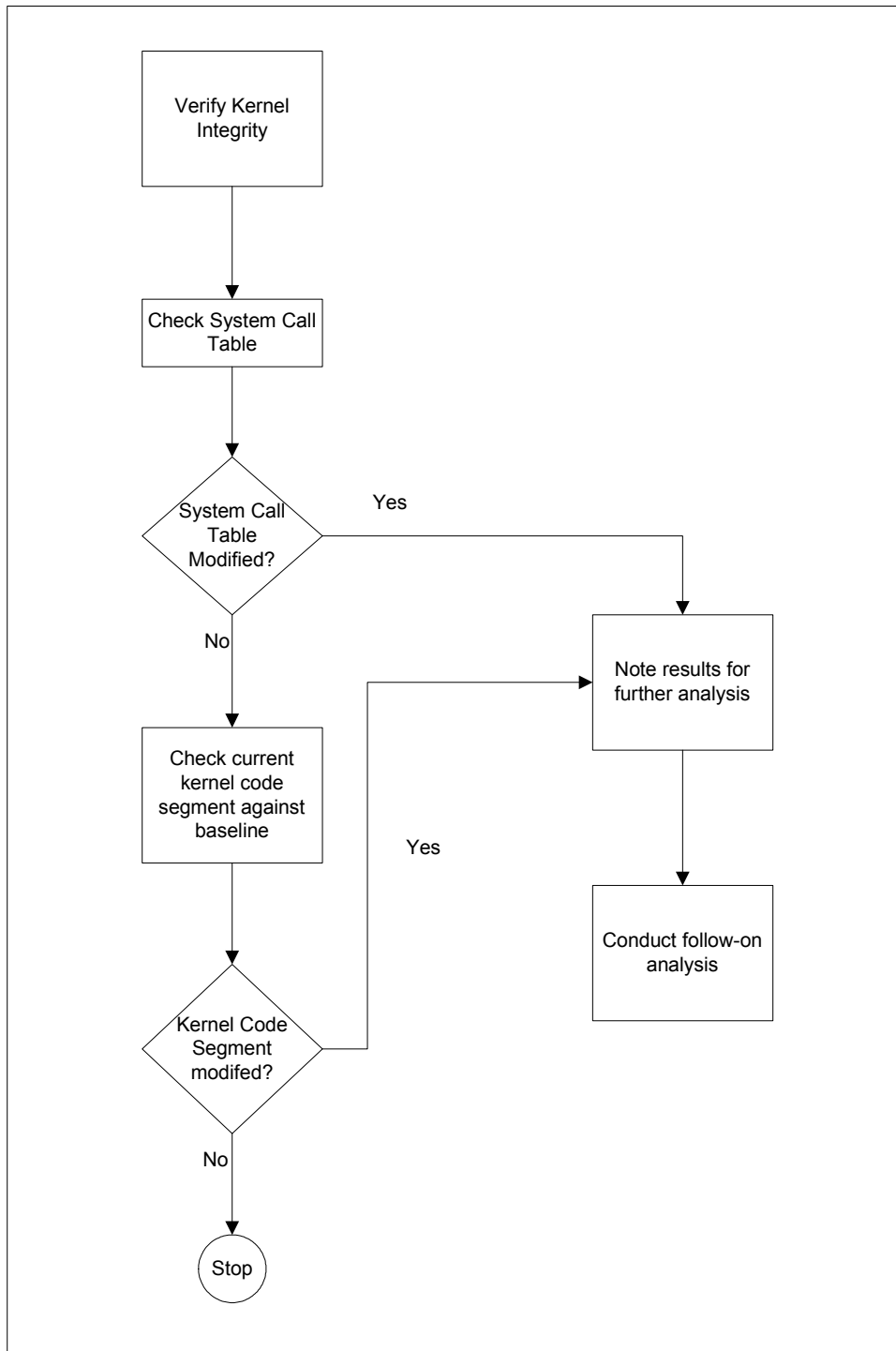
To detect a rootkit of this type a methodology similar to what the hacker used must be followed. Our detection method queries the same register that is targeted by the rootkit described above. Using this information we are able to retrieve the address of the system call table that is currently in use on the target system. The addresses of the system calls that are currently in use on the target system can then be compared against the addresses of the system calls in the `/boot/System.map` file in a manner similar to that described above to detect the presence of a kernel level rootkit that modifies system calls. A mismatch between the address of the system calls from the `/boot/System.map` file and the addresses that are listed within the current `sys_call_table` from kernel memory may be an indication that the kernel has been compromised by a rootkit that targets system calls. It

is significant to note however, that the version of the System.map file used for comparison must correspond to the current kernel installation on the target system. A new System.map file is created whenever the kernel is recompiled. Using an outdated System.map file will most likely result in mismatch with the sys\_call\_table that is resident within kernel memory.

Analysis may reveal that the system calls have not been modified. The rest of the kernel code segment should be analyzed for possible exploitation by a kernel level rootkit. This can be accomplished by comparing the current segment of kernel code that resides within kernel space against the version of the kernel code that we archived for analysis. Kernel code was archived for analysis when we established a baseline for this system prior to infecting the system with the rootkit. A difference between these two files will indicate that some other area of the kernel may have been targeted for exploitation by the rootkit. A GPL program such as chkidt, described in section 3.1, can be used to check the integrity of the Interrupt Descriptor Table. A comparison of the currently running kernel can also be accomplished by kdb to disassemble addresses where the particular differences occur.

If this analysis indicates that no modifications to the system calls have occurred and the current segment of kernel code matches the baseline segment of kernel code then we assume that the kernel has not been modified by this rootkit. Analysis can focus on the system level utilities that have been detected as being modified. Segments of the kernel that may have been modified must undergo further analysis to identify a possible  $\nabla$  for classification. It is significant to note that a rootkit may contain elements that modify both the kernel and system utilities on the target system.

The following figure shows the steps that must be taken to verify the integrity of the kernel and to establish that a kernel rootkit may be present.



**Figure 24: Verify kernel**

#### **4.2.13 Analyze results of Kernel Integrity Check**

The kernel integrity checker program `kern_check` will detect modifications to the system call table. This is an indication that the corresponding system calls within kernel space have been modified by the rootkit. The `kern_check` program will output those system call addresses currently listed in the active `sys_call_table` that do not match the system call addresses listed in the `System.map` file. If the current `System.map` file is up to date with the current kernel, then the `system_call_table` has been modified by a kernel rootkit. Methods to intercept system calls have been published and are available on the Internet [42].

A program such as `StMichael`, discussed in section 3.2, would be detected in this manner since it is installed via a loadable kernel module and changes four system calls along with their underlying addresses. Any system call can be redirected by a loadable kernel module in the current versions of the Linux kernel. Since we are initially starting with a clean operating system install any loadable kernel module installations are unexpected and suspicious in nature. System calls can also be modified via `/dev/kmem`. These are the areas that are to be analyzed for identifying potential  $\nabla$ 's for follow-on characterization.

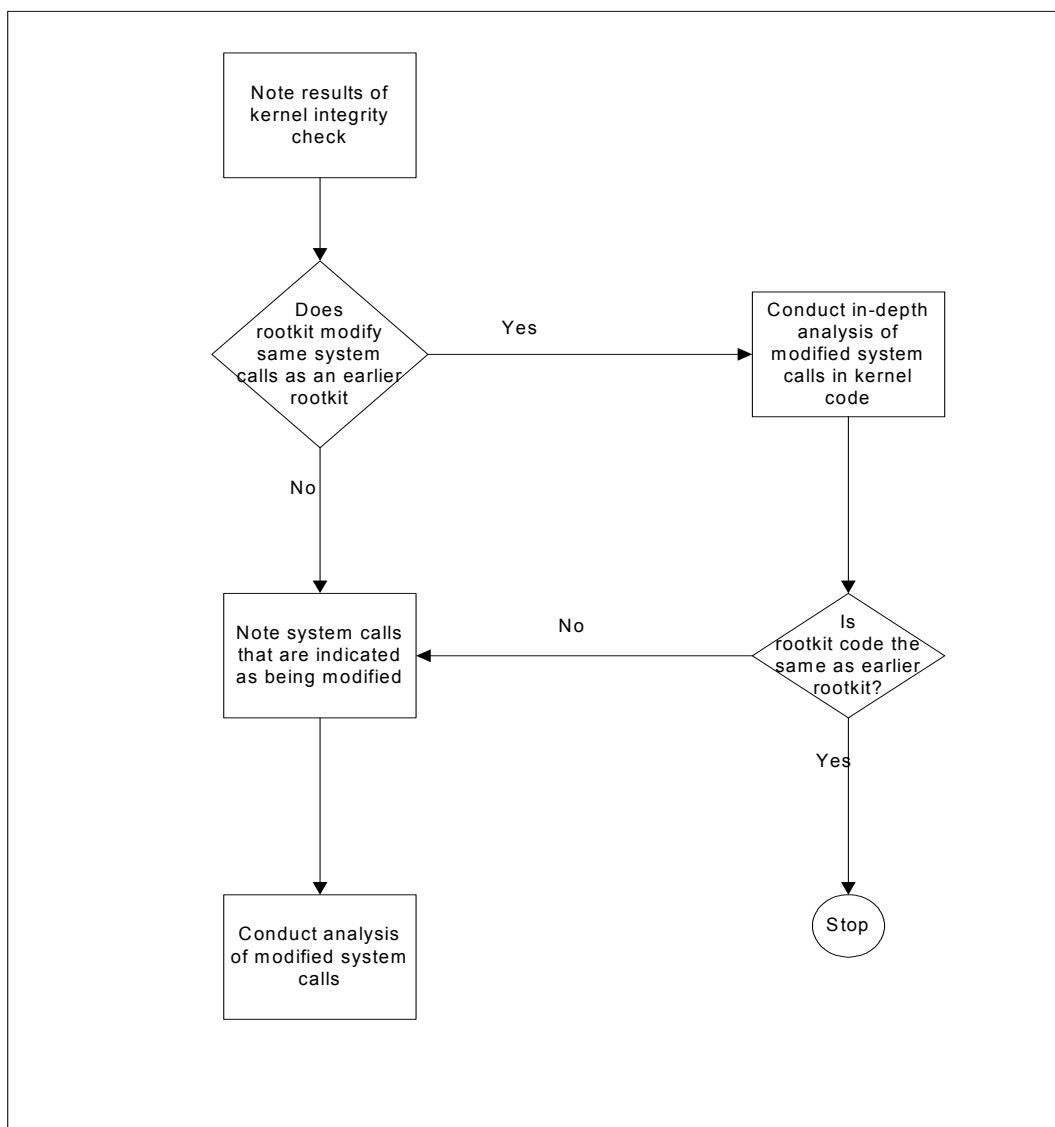
Many of the kernel rootkits that have been analyzed thus far that target system calls make use of the original system calls which are renamed. This is done to maintain the functionality of the original system calls which is to be incorporated into the modified system call [42]. Kernel rootkit syscalls wrap the original system call address within their exploit code.

It may be possible to identify a particular kernel rootkit by documenting the particular



system calls that the rootkit modifies [34]. These modified system calls can be detected by the above analysis. Two separate kernel rootkits that modify the same set of system calls may require more detailed analysis. One of these rootkits may just be a modification of the other rootkit.

The following figure shows the analysis that is performed on the results of the kernel integrity check.



**Figure 25: Kernel Integrity Check Analysis**

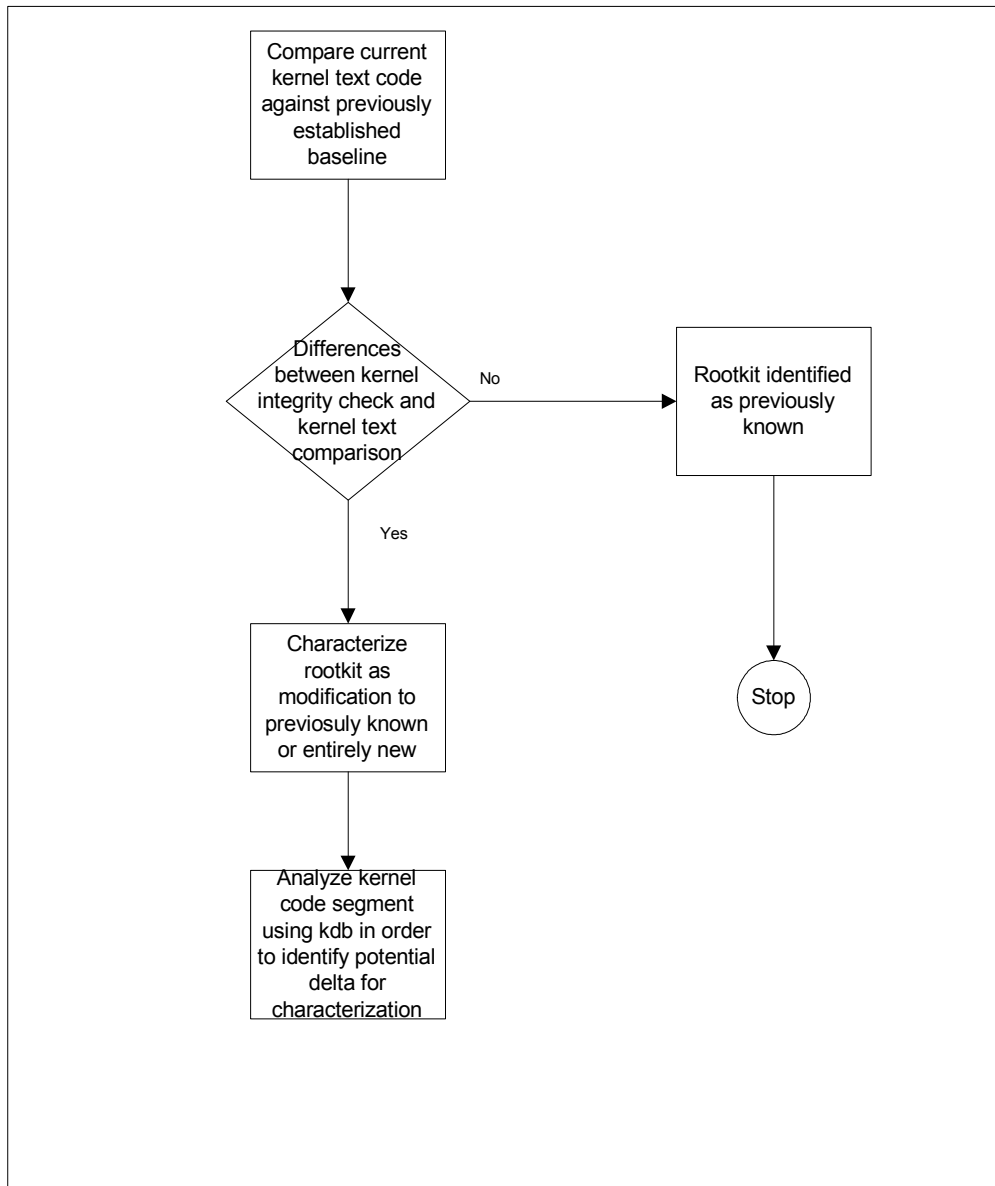
#### **4.2.14 Compare current kernel text code against baseline**

A kernel rootkit may attempt to overwrite the valid address space of the system call with modified system call code while maintaining the original system call functionality. Theoretically, it may be possible to move the original system call address space to some other portion of kernel memory and access it from the original address. This would still be detectable from the baseline kernel code comparison. However, we have not examined any kernel rootkits that follow this methodology in conducting this research. Any changes to the kernel not detected by previous analysis should be detected by comparison of the kernel text code that currently resides in memory with the baseline kernel text code that was previously established.

At this point a comparison must be made between the results of the kernel integrity check and the kernel text code comparison. The system calls that are detected as being modified by the kernel integrity check program should also be detected as being modified by the kernel text code comparison. If there are no additional modifications to the kernel text code then the rootkit is most likely a previously known rootkit. Additional modifications to the kernel text code that were not detected by the kernel integrity check are indications that the rootkit under analysis is either a modification to an existing kernel level rootkit or an entirely new kernel level rootkit.

The kernel code segment for the potential modification to an existing rootkit or entirely new rootkit is to be analyzed to classify this rootkit. The kdb kernel debugger can be used to analyze the segments of kernel text code that are indicated as having been modified as a result of the comparison with the original kernel text code segment from the baseline that was previously established. The *md* and *id* commands can be used to

examine the kernel memory as well as to disassemble segments of kernel code for follow on analysis. The following figures shows the analysis that is performed on comparing the results of the kern integrity check against the kernel text comparison to determine if we are dealing with an existing, modification to existing, or entirely new rootkit.



**Figure 26: Comparison of current kernel text code**

### 4.3 Summary

We have proposed a methodology for the detection and classification of rootkit exploits as either existing, modification to existing, or entirely new. This framework seeks to identify a delta ( $\nabla$ ) within the rootkit exploit that is to be used in the classification process.

A methodology was then proposed to identify data to use for classifying rootkit exploits. We presented steps that were to be taken during the classification process. We made use of current GPL detection methodology during this analysis and conduct manual analysis when necessary. Application of this methodology to a compromised system result in the identification of the specific elements of the compromised system that are to undergo further analysis for classifying the rootkit.

## Chapter 5

### New Methods to Detect and Classify Rootkit Exploits

The purpose of this chapter is to describe methods that can be used to detect and classify certain types of rootkit exploits. We have already presented a formal methodology to classify rootkit exploits as existing, modification to existing or entirely new in the previous chapter. A key component of our methodology is the ability to identify a delta ( $\nabla$ ) within the rootkit exploit. This  $\nabla$  is some identifiable characteristic between the rootkit and the underlying program or capability that the rootkit is intended to replace on the target system. The formal methodology to classify rootkit exploits will indicate potential areas for further analysis to identify a  $\nabla$  that can be used for both classification and follow-on detection of the rootkit. We propose methods that can be used to identify this  $\nabla$  characteristic. These methods will address rootkits that replace system utilities as well as those that target the underlying kernel of the target operating system. We also present changes that were made to the chkrootkit program as a result of our methodology.

The vulnerabilities that exist in modern operating systems as well the proliferation of exploits that allow hackers to gain root access on networked computer systems provide hackers with the ability to install rootkits on systems once root access has been acquired. System administrators need to be aware of the threats that their computers face from rootkits as well as the ability to recognize if a rootkit has been installed on their computer system.

## 5.1 Detection of Unique String Signatures in Binary System Utility

### Exploits

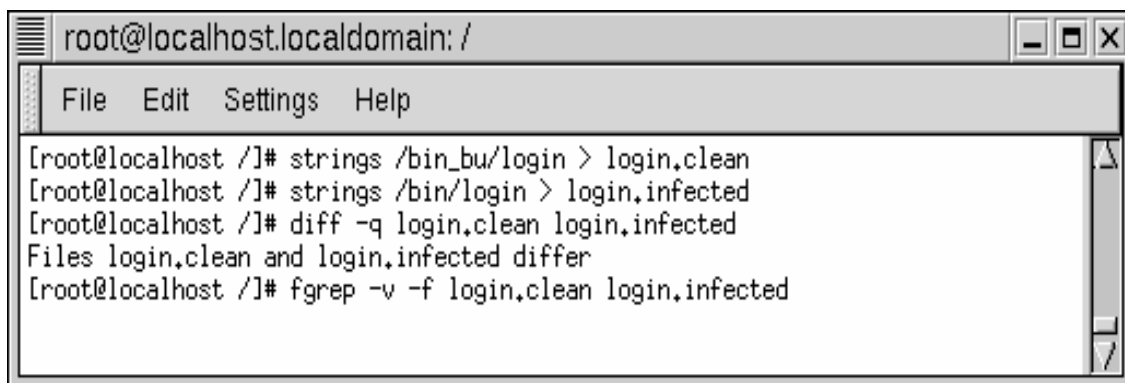
We propose a methodology to uniquely identify the different rootkits that target binary system utility programs for replacement. The Linux Rootkit IV (lrk4) developed by Lord Somer is an example of this type of rootkit. To accomplish this we will make use of the print lines matching a pattern (*fgrep*) utility that is available on the target system. It is best to use a version of *fgrep* that was archived from the target system prior to that system being infected with the rootkit to ensure that the version of *fgrep* that is in use was not a target of the rootkit. It is necessary to have a clean copy of each binary file that was replaced by the rootkit program. The listing of the files that were replaced would be available as a result of running AIDE on the target system as discussed in the formal methodology. Copies of the infected binary files are available on the target system. For example, on a target Red Hat 6.2 system running the Linux 2.2 kernel, the login program is one of the program files that is indicated as having been changed by the AIDE program after the lrk4 rootkit was run on this system. A clean login binary exists in the bin\_bu directory, which was created when the operating system was first installed based on the documentation that was available with the lrk4 rootkit. The lrk4 documentation indicates that the various system utilities maintained in the /bin directories are targets for replacement by the lrk4 rootkit. Results from the program AIDE indicates the login program in the /bin directory is infected by lrk4 indicated by the AIDE program. The methodology to identify unique  $\nabla$  characteristics is as follows:

1. Run the *strings* command on each file in question and pipe the results into a file for further comparison.

2. As an additional check run the *diff* command against these two files for a check to see if the strings contained in the two files are different. Use the *-q* switch so that the output only reflects if the files are different.
3. Run the following command: *fgrep -v -f login.clean login.infected*

The *fgrep* command outputs a line-matching pattern.

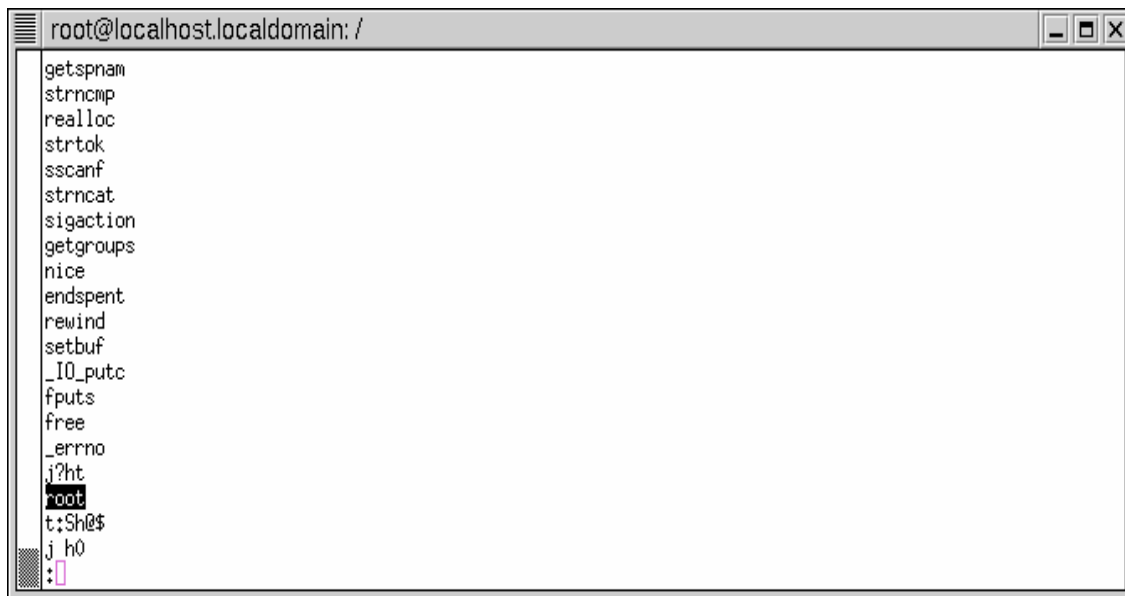
The *-v* switch is an invert-matching switch which tells the *fgrep* function to only output those lines that do not match. The *-f* switch tells the *fgrep* command to get the patterns to use for matching in the second file (*login.infected*) from the first file (*login.clean*). Figure 27 shows this series of commands being executed.

A terminal window titled 'root@localhost.localdomain: /' with a menu bar (File, Edit, Settings, Help). The terminal shows the following commands and output:

```
[root@localhost /]# strings /bin_bu/login > login.clean
[root@localhost /]# strings /bin/login > login.infected
[root@localhost /]# diff -q login.clean login.infected
Files login.clean and login.infected differ
[root@localhost /]# fgrep -v -f login.clean login.infected
```

**Figure 27: commands to compare login files**

This series of commands compares the strings that exists in both files and outputs only those existing strings that are different between the two files. Using the clean login file as the string source and the infected login file as the target file will result in the output of those strings that exist in the hacked version of the login program but do not exist in the clean login file. The output of this command is displayed in the Figure 28 with the string 'root' highlighted.



```
root@localhost.localdomain: /
getsnam
strncmp
realloc
strtok
sscanf
strncat
sigaction
getgroups
nice
endspent
rewind
setbuf
_IO_putc
fputs
free
_errno
j?ht
root
t:Sh$
j h0
:
```

**Figure 28: Output of fgrep function**

There are numerous strings that differ between the clean and hacked login file. The primary reason for this is that analysis of the lrk4 source code which was available showed that the lrk4 login program is based on the Shadow-Suite login.c code and the clean login program is based on the BSD login.c code. However, some of these strings are potential  $\nabla$  characteristics that can be used to detect and classify this rootkit. These  $\nabla$  characteristics can be used as signatures for the chkrootkit program to use to check for the existence of an infected login program. The fact that we had the clean login.c code from the Shadow-Suite made it easy to determine what code, to include what strings, had been added by Lord Somer. All that is then required is to check these added strings against the original clean login.c binary file to ascertain the validity of using these  $\nabla$  characteristic strings as a signature.

By using our methodology a system administrator could build a library of files that may contain the text strings that exist in the system binaries. Even if these files do not



contain any unique strings they can still serve as a unique signature for a specific rootkit. As various rootkits are discovered additional unique binary files can be added to the library. A system administrator that determined that a binary system utility rootkit had been installed on a system could follow our methodology to compare the infected system binaries with the files that exist in the library.

Thus, if an infected binary did not match with the existing binaries in the library, the system administrator could make the determination that the system has been infected with a new or modified rootkit since it does not match any of the existing signature files.

The text strings that exist in this new or modified rootkit can be examined for unique strings to identify this new trojan exploit. This unique  $\nabla$  characteristic string could be used by the chkrootkit program to identify this specific rootkit exploit. A common  $\nabla$  characteristic text string could also be sought so that the chkrootkit program would be able to detect the greatest number of exploits with the least number of signatures. In either case, this  $\nabla$  characteristic signature can also be provided to a signature-based IDS system for detection of this exploit.

## **5.2 Modifications to the chkrootkit program**

We initially installed the Lord Somer's lrk4 rootkit on a clean Red Hat 6.2 system. We then ran chkrootkit-0.36, which was the current available version of chkrootkit, against the system. This version of chkrootkit detected that some of the binaries had been infected, but it did not detect that the login binary had been infected. The lrk4 rootkit that we installed did contain a source login.c program with a trojan capability as previously discussed in the last section.

Upon analysis, we discovered an error in the logic of the chkrootkit program. The

chkrootkit suite is a script called chkrootkit that calls a routine called chk\_login. This routine performs signature analysis on the login program by looking for the appearance of various strings within the binary file. One of the strings used by the chkrootkit program to detect infected login programs is the string “root”. The lrk4 login binary file has 2 instances of the string “root” within it. The clean login program does not contain any reference to the string ‘root’. The chk\_login routine was written to allow for the appearance of 2 or less instances of “root” in the login binary program. We contacted Nelson Murilo, who is one of the authors of the chkrootkit program, about our discovery and provided him with the results of our analysis. The chkrootkit code was modified to only allow for the appearance of the string “root” in the login file for those specific operating systems that have the string “root” appear in the clean version of their login files. A new version of the chkrootkit program, chkrootkit-0.37, was quickly released that detected that the lrk4 login file is infected.

### **5.3 Detection of Unique String Signatures in Binary System Utility**

#### **Exploits using String Hiding Techniques**

A rootkit developer may choose to use string hiding techniques to defeat the signature analysis checks for rootkit exploits. It may be possible for the developer of a rootkit exploit to create a binary trojan file without any distinct differences in the string signatures between the trojan file and the original good binary file. One method to accomplish this is to use a character array with the rootkit source code to store string values. If the developer were to use a character array to hide the trojan password in the file than it may not be possible to detect the trojan password with the *strings* command. The normal method to write character strings into an array is as follows: *strcpy* (rewt,

“rewt”). This would result in the string rewt being visible within the compiled program. Usernames, passwords, or any other text string written individually into an array will only appear in the binary files as a sequence of single characters separated by instructions as opposed to a sequence of adjacent characters. According to the MAN page for the *strings* command, the default search setting is to search for text strings of at least 4 characters in length. This value can be reset to a smaller value. However, if a character array is used and individual characters that are written into an array one character at a time then the results would be different at compile time. Each individual character would most likely be surrounded by OPCODE instructions and memory and register locations. The binary executable program that is generated by the compile would not have any string values within it. This binary executable is examined by the chkrootkit program for specific string signature values.

Running the file integrity check on a system with an installed rootkit may indicate that one or more of the system utility binary files has been altered. The chkrootkit program, which is primarily string signature based, may not detect this rootkit installation if string hiding techniques are being utilized. Conducting an analysis of these binary utility files with the methods we have previously discussed for detecting unique string signatures may not result in the identification of any unique signatures in the modified files. Text strings that are necessary for proper rootkit execution may be hidden with string hiding techniques within the rootkit program.

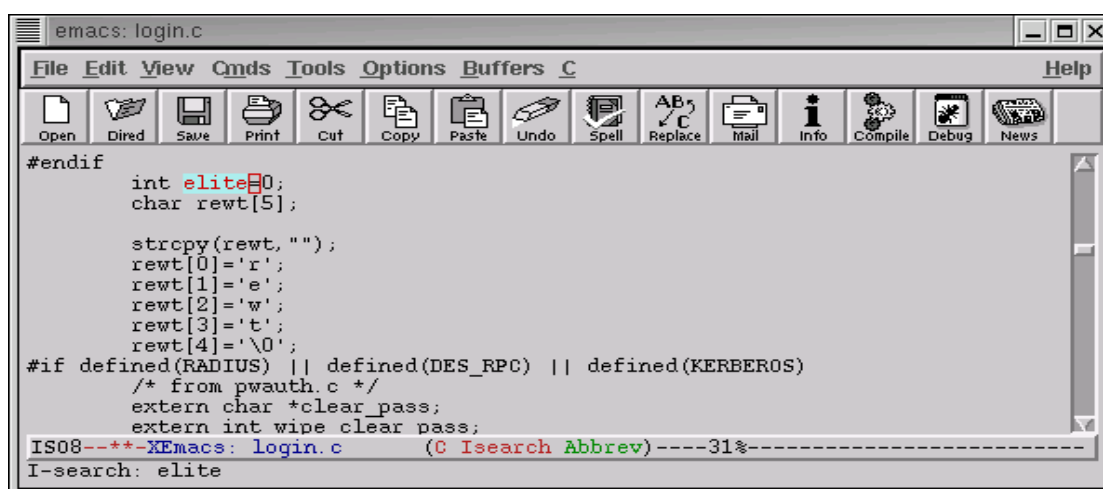
We recognized that a method needed to be developed for identifying rootkit text strings hidden in character arrays within binary files. We present an examination of the binary code of a trojan login module as a case study to identify string hiding techniques that may

be used by hackers in the development of trojan login programs. The login program is a logical choice for string hiding techniques to hide the trojan password or the trojan username and password combination. Lord Somer, in lrk4, makes a reference to the use of string hiding techniques in the rootkit.h file, which is the header file for the lrk4 rootkit exploit. The fact that Lord Somer references string hiding techniques indicates that this technique may be in use by some hackers in the development of rootkit exploits. Text strings that are hidden in this manner may be used to characterize and identify specific rootkits if these text strings can be identified and extracted from the trojan binary code if the rootkit source code is not available. If the source code is available then we can look for strings written into character arrays.

When we began analyzing binary files for hidden strings we recognized that we needed some type of program that is capable of viewing these types of files. The program that we chose to utilize is BIEW. This GPL program is described as follows: “BIEW (Binary vIEW) is a free, portable, advanced file viewer with built-in editor for binary, hexadecimal and disassembler modes” [43]. This is in keeping with our goal to use open source tools to conduct our analysis.

Analysis of the lrk4 source code for the login program indicated that the term ‘rewt’ was stored within a character array called rewt. The name “rewt” is the hacker’s modification on the word root. If the source code is available for a rootkit, efforts to use string hiding techniques can be identified by conducting an in-depth analysis of this source code. Any character arrays within the source code can be examined to determine their use within the rootkit program. A check can be made against the original source code that the rootkit is intended to replace to see if this same character array exists. If

this character array does not appear in the original source code and the string value that it contains is used in a comparison with some value that is input by a user then this array and its associated value would be suspicious in nature. The following figure shows an example screen display of this type of character array that was found in the lrk4 login source code. This character array does not appear in the original login source code.



```
#endif
    int elite=0;
    char rewt[5];

    strcpy(rewt, "");
    rewt[0]='r';
    rewt[1]='e';
    rewt[2]='w';
    rewt[3]='t';
    rewt[4]='\0';
#if defined(RADIUS) || defined(DES_RPC) || defined(KERBEROS)
/* from pwauth.c */
extern char *clear_pass;
extern int wipe clear pass;
IS08--*-XEmacs: login.c (C Isearch Abbrev)----31%-----
I-search: elite
```

**Figure 29: rewt array used for sting hiding**

The name “rewt” is the hacker’s modification on the word root. If the source code is available for rootkit efforts to use string hiding techniques can be identified by conducting an in-depth analysis of this source code. However, the source code for the rootkit exploit may not always be available. We set out to find a methodology to detect text strings hidden within character arrays for systems utilizing the Intel i386 architecture, which is the common platform of choice for running the Linux operating system. This methodology may be transferable to other architectures and can be modified to identify other array types.

Both the source and binary code for the clean and hacked version of the login routine

are available for examination. Running the find differences between two files (*diff*) command on the two source files outputs the differences between the files. The first noticeable difference is that an integer variable ‘elite’ and a five-character array “rewt” are introduced into the login.c program. We speculate that the name “rewt” is the hacker’s modification on the word root. A common hacker signature is 31337. 31337 spells out “eleet” in the dialogue used by some hackers [44]. We believe that the “elite” variable is the hacker’s modification of eleet. This integer value is initially set equal to zero. The five-character array rewt stores the name “rewt” and a terminating null symbol.

We then searched the binary source file for the name rewt to determine if we could find a way to recognize this term. We conducted a manual analysis of this file by searching for the individual characters ‘r’, ‘e’, ‘w’, and ‘t’ in sequence within the binary file using the bmove program within the BIEW tool set. This tool set is available on the web at: <http://sourceforge.net/projects/biew>. The following screen shows where we found the term ‘rewt’ within the binary lrk4 login program. The characters ‘r’ (ASCII value 72 hex), ‘e’ (ASCII value 65 hex), ‘w’ (ASCII value 77 hex), and ‘t’ (ASCII value 74 hex) appear starting at the second line of the display.

The screenshot shows a hex editor window titled 'root@localhost.localdomain: /bvi-1.3.1'. The menu bar includes 'File', 'Edit', 'Settings', and 'Help'. The main display area shows a hex dump of memory. The second line of the display (address 00001C70) contains the characters 'r', 'e', 'w', and 't' in sequence, which corresponds to the ASCII values 72, 65, 77, and 74 hex. The characters are preceded by a null terminator (00). The rest of the line and subsequent lines contain various other characters and null bytes. At the bottom of the window, a status bar indicates '--More--(17%)'.

Address	Hex	ASCII
00001C60	00 00 00 8D 8D 58 DD FF FF 89 8D 34 DD FF FF C6	.....X.....4....
00001C70	85 58 DD FF FF 72 C6 85 59 DD FF FF C6 85 5A	.X...r..Y...e..Z
00001C80	DD FF FF 77 C6 85 5B DD FF FF 74 C6 85 5C DD FF	...w...[...t...\..
00001C90	FF 00 E8 81 13 00 00 C6 45 E0 00 E8 8C FB FF FF	.....E.....
00001CA0	89 C2 85 D2 0F 94 C0 88 85 20 DD FF FF 88 C1 83	.....<\$.....>u.
00001CB0	E1 01 89 0D 3C 24 05 08 BE 01 00 00 00 3B 75 08	.....<\$.....>u.
00001CC0	7D 2D 8B 5D 0C 83 C3 04 8B 3B 80 3F 2D 75 17 31	}-..]......;?-u.1
00001CD0	C0 FC B9 FF FF FF FF F2 AE F7 D1 49 83 F9 02 76	.....I...v
00001CE0	05 E8 52 FE FF FF 83 C3 04 46 3B 75 08 7C D9 8B	..R.....F;u..l..
00001CF0	4D 0C FF 31 E8 5F 10 00 00 89 C2 89 15 38 24 05	M..1.....8\$.
00001D00	08 83 C4 04 E9 B4 00 00 00 8D B4 26 00 00 00 00	.....&.....
00001D10	83 C2 9C 83 FA 0E 0F 87 9C 00 00 00 FF 24 95 50	.....\$.P
00001D20	F5 04 08 FF 05 AC 8F 05 08 E9 8F 00 00 00 89 F6	.....

Figure 30: string 'rewt' written into character array

Each of the four ASCII characters is in a sequence of seven bytes with the last byte being the value of the ASCII character that is to be written into the character array. The first byte in each seven byte series is the opcode 'C6'. Since our research involves the Linux operating system running on Intel x86 Pentium processors, we went to the Intel Embedded Pentium Processor Family Technical Information Center at <http://www.intel.com/design/intarch/techinfo/pentium>. According to the opcode map available at this site, the 'C6' opcode corresponds to the MOV mnemonic [45]. A separate description of the MOV command is also available at this site. This command is described as follows: Move immediate one byte value (the ASCII character) to a one byte register/memory location. This is the exact behavior that we would expect for a command to store a single character from a string into a character array.

It may also be possible to identify strings being written into a character array by disassembling the binary file since both disassembly and analysis by the BIEW program provide a one to one mapping of the hex code within the binary file. The BIEW program does provide you with printable character output which makes it easier to search for the particular characters that compose the string.

Now that we have a particular mnemonic command (MOV) and opcode associated with this particular string hiding technique. We can then use this information to search binary files for instances of strings being hidden into character arrays. A search can be conducted for all consecutive instances of the C6 opcode command separated by a consistent number of bytes capturing the last byte. This last byte is the ASCII character that is being written into a character array. All of the individual bytes collected from a consecutive series of C6 opcodes can be concatenated together to form a string.

Characteristics of this string (e.g. length) can be used as a signature for this rootkit exploit. This same methodology could also be followed to identify data that is to be written within other types of data structures.

## **5.4 Detection of Kernel Rootkit Exploits by Examination of System Call Table Entry Point in the Kernel**

Checking the System Call Table in kernel memory against the `/boot/System.map` file has already been proposed. This is the technique that the Samhain program `kern_check` utilizes to detect for instances of kernel level rootkits. However, the original `kern_check` program fails to detect rootkits of the System Call Table redirection variety as well as to detect any type of rootkits on more recent versions of the Linux kernel.

Our examination of the SuckIT rootkit which is a rootkit that redirects the system call table, revealed to us the first difference, or  $\nabla$  in functionality between SuckIT and the program that it replaces. SuckIT overwrites a location in kernel memory that contains the address of the system call table. SuckIT is able to accomplish this by querying a specific register within the processor. It then uses this information to find the entry point address within the kernel for the system call table and overwrites this address with the address of a new system call table containing the addresses of some malicious system calls that SuckIT also creates. We present an in-depth analysis of how SuckIT accomplishes this within the appendix of this thesis.

We now have a  $\nabla$  consisting of a redirected system call table address, a new system call table, and some new malicious system calls. We propose that you can use the same method that SuckIT uses to query the processor to retrieve the address of the system call table to check and see if this address has been changed by a rootkit such as SuckIT. The



original address is available when the kernel is first compiled and this address is stored in the `/boot/System.map` file. If these addresses differ then a more detailed check can be made of the system call table that currently exists in kernel memory to develop a  $\nabla$  between the addresses of the system calls that exist in system call table within kernel memory and the addresses of the system calls that exist in the `/boot/System.map` file.

If the `/boot/System.map` file is current then differences between it and the system call table within kernel memory will indicate that redirection of the system calls is occurring on the system and that the system is infected with some type of rootkit. A preliminary signature can be established based on the number of system calls that are being redirected on the target system. If two different kernel level rootkits change a different number of system calls then we can assume we have two different kernel level rootkits. If these two rootkits change the same system calls then we can conduct a more detailed analysis of each infected system to look for differences between the two rootkits.

This method will also work in the detection of kernel rootkits that modify the existing system call table in the more recent versions of the Linux operating system. We are now able to query the system for the address of the system call table that is currently in use by the kernel. This system call table can then be analyzed for exploitation.

If we do not have the rootkit source code available we can still look for differences though either the `kdb` program or we can copy segments of kernel memory through `/dev/kmem` and examining this data off-line. We can use `kdb` to examine the actual machine code of the malicious system calls since we will have the actual addresses of these malicious system calls within kernel memory. We can also try and disassemble these malicious system calls manually or through the `kdb` program if it is installed on the

system that we are using to investigate this kernel level rootkit.

In any case, we are now able to detect that redirection of the system call table is occurring on the target system. We do realize that a hacker may be able to develop a kernel level rootkit that could provide false information concerning the entry point of the system call table within the kernel. At present, however, we are unaware of any kernel level rootkit that is able to do this.

## 5.5 Summary

In this chapter we have proposed some methods for identifying a specific  $\nabla$  characteristic within the rootkit exploit that is to be used in the detection and classification process. Examination of an existing rootkit should result in a  $\nabla$  that is already known and examination of a modification to an existing rootkit should result in a  $\nabla$  that is similar to the  $\nabla$  for an already known rootkit. An entirely new rootkit exploit should result in a  $\nabla$  that has characteristics that has never been seen in a previously examined rootkit exploit.

Our analysis involved rootkits that replace system utilities as well as those that target the underlying kernel of the target operating system. The methods that we presented in this chapter will allow a researcher to identify a specific  $\nabla$  for the rootkit that is being examined.

## **Chapter 6**

### **Establishment of a Honeynet to Detect New Rootkit Exploits**

We have previously expressed that a key component of our methodology is that we require a copy of the rootkit that we wish to detect and characterize. The method that we propose to use to be able to do this is to use a Honeynet. An added benefit to using a Honeynet to collect research data is that the Honeynet can improve the overall security of the network where it is employed [46]. We believe the use of a Honeynet within a network can provide additional network security. The Honeynet can serve as a complement to the use of the firewall and IDS and help to overcome some of the shortcomings that are inherent to these systems. The Honeynet will also allow for the collection of any rootkits that are targeted against any systems on the Honeynet.

#### **6.1 Definition of a Honeynet**

A Honeynet is a network, placed behind a reverse firewall that captures all inbound and outbound data. The reverse firewall limits the amount of malicious traffic that can leave the Honeynet. This data is contained, captured, and controlled. Any type of system can be placed within the Honeynet, to include those systems that are currently employed on the network that the Honeynet is intended to protect. Standard production systems can be used on the Honeynet to give the hacker the look and feel of a real system. A Honeynet is a network intended to be compromised so as to provide the system administrator with intelligence about vulnerabilities and compromises within the network [47].

There are two critical principles concerning the successful operation of a Honeynet. These two principles are the concept of Data Capture and Data Control. Both of these principles must be followed for the Honeynet to be successfully employed.

The principle of Data Capture concerns information gathering. All information that enters or leaves the Honeynet must be collected for analysis. This data must be collected without the knowledge of the individuals who are conducting malicious activity against the network that is to be protected. This is to prevent the hacker from bypassing the Honeynet network. The data that is collected must be stored in a location different from the Honeynet. This is done so that if the hacker compromises a Honeynet system, the data cannot be destroyed or altered. The goal is to be able to capture data on the hacker without the hacker knowing that this data is being collected.

The principle of Data Control concerns protecting other networks from being attacked and compromised by computers on the Honeynet. If a hacker compromises a Honeynet system, then this hacker must be prevented from using this system to attack and compromise production systems on other networks. The process of Data Control must be automated to prevent the hacker from getting suspicious. We do not want the hacker to become aware of the fact that the system compromised is on a Honeynet [48].

## **6.2 Honeynet Establishment**

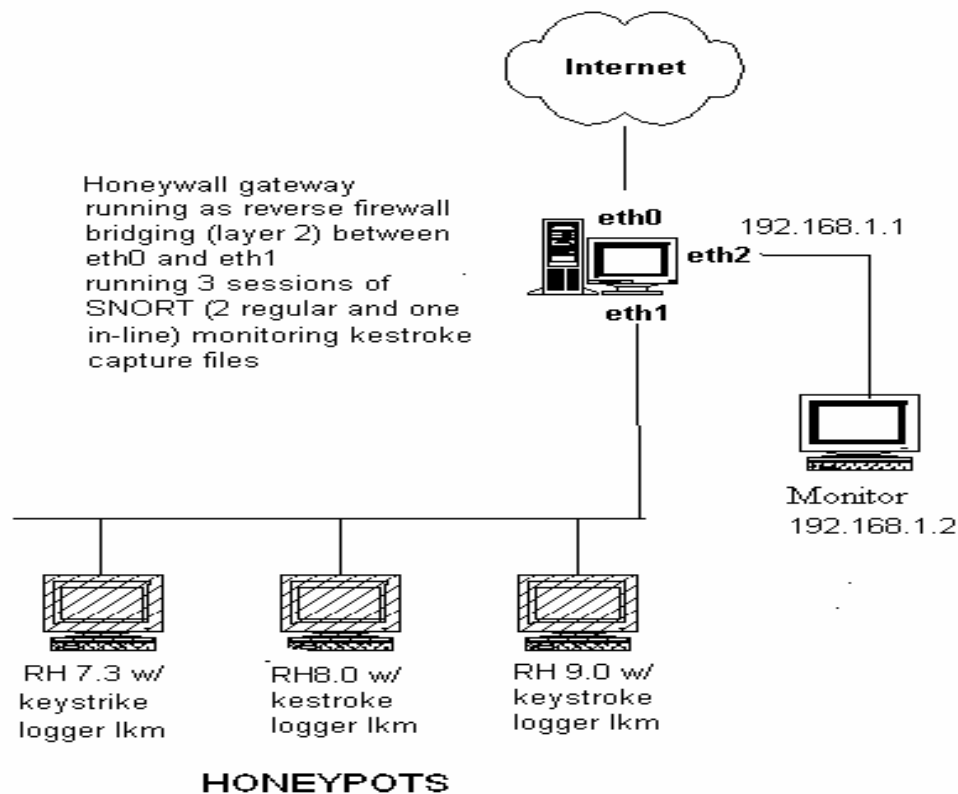
There are currently two types of Honeynets that can be employed on a network. These are GEN I, or first generation, and GEN II, or second generation. The type of Honeynet that one chooses to use depends on many factors to include availability of resources, types of hackers and attacks that you are trying to detect, and overall experience with the Honeynet methodology.

GEN I Honeynets are the simpler methodology to employ. This technology was first developed in 1999 by the Honeynet Alliance. Although GEN I Honeynets are somewhat limited in their ability for Data Capture and Data Control, they are highly effective in detecting automated attacks or beginner level attacks against targets of opportunity on the network. Their limitations in Data Control make it possible for a hacker to fingerprint them as a Honeynet. They also offer little to a skilled hacker to attract them to target the Honeynet, since the machines on the Honeynet are normally just default installations of various operating systems.

GEN II Honeynets were developed in 2002 to address the shortcomings inherent with GEN I Honeynets. The primary area that was addressed by GEN II Honeynets is in the area of Data Control. GEN I Honeynets used a firewall to provide Data Control by limiting the number of outbound connections from the Honeynet. This is a very effective method of Data Control, however, it lacks flexibility and allows for the possibility of the hacker fingerprinting the Honeynet. GEN II Honeynets provide data control by examining outbound data and making a determination to block, to pass, or to modify by changing some of the packet contents so as to allow data to appear to pass but rendering it benign. GEN II Honeynets are more complex to deploy and maintain than GEN I Honeynets [49].

We had chosen to initially deploy a GEN I Honeynet on our enterprise network. We were concerned with detecting machines within our enterprise network that had been compromised by automated script type attacks in addition to collecting rootkit research. We later employed a GEN II Honeynet in our research efforts.

The Georgia Tech Honeynet was initially established during the Summer of 2002. It was established using open source software and with equipment that is not currently state of the art. Initially, it was established as a single computer but was expanded to include several different machines running various operating systems. The following diagram (Figure 31) shows a typical configuration of the GEN II Georgia Tech Honeynet.



**Figure 31: The Georgia Tech Honeynet**

An IP address range was provided to us by the Georgia Tech Office of Information Technology (OIT) to establish this Honeynet. This block of addresses is within the address range that belongs to Georgia Tech campus network. This address range is accessible by both the Georgia Tech campus network and the Internet.

As previously stated, the hardware that was used to establish the Honeynet was not current state of the art equipment since the machines running on the Honeynet have no production value. The amount of traffic going to and from the Honeynet should be minimum since these systems are not running any production software. The system that runs as the Firewall does only that, it has no other applications running on it. We used the LINUX operating system on the firewall. We ran Red Hat version 8.0 utilizing a custom Linux 2.4 kernel (customized to allow for bridging). Therefore, it was entirely possible to set up a Honeynet on a network using surplus equipment that may be available within the enterprise.

Although there are commercial versions of software and products available to establish a Honeynet, we chose to establish the Georgia Tech Honeynet using Open Source Software. This is keeping with our goal of using open source software within our methodology and open source software provides us with the greatest flexibility.

We used the rc.firewall script developed by The Honeynet Alliance to set up our firewall and establish Data Control for our Honeynet. This script is available from The Honeynet Alliance [50]. The purpose of this script was to perform Network Address Translation (NAT) for the target machines on the GEN I Honeynet. Currently, this script is functioning in a layer 2 bridging capacity. The Honeynet firewall does not have a Internet routable IP addressed assigned and bridges all data from the Internet to the Honeynet at layer 2. For both GEN I and GEN II Honeynets the script provides data control.

The rc.firewall script was modified to control the Honeynet systems by restricting the number of outbound connections that are allowed from target systems on the Honeynet

This script also has the capability to pass, drop, or modify outbound data packets depending on the specific rule set that is implemented. This script should work with any version of LINUX.

The IDS that we chose to use to monitor the Honeynet is SNORT. SNORT is open source IDS software [51]. SNORT is primarily signature based but does have an anomaly detection plug-in available. Signatures are available periodically from the SNORT website and it is possible to write your own signatures. The IDS runs on the Honeynet firewall which is isolated and not accessible from the Honeynet network. The network monitoring system is currently running Red Hat 8.0 software. For the GEN I Honeynet the system utilized a network interface card (NIC) set in the promiscuous mode in a hub that connects all of the computers on the Honeynet. This NIC card did not have an IP address assigned to it so that a hacker on the Honeynet would not have visibility of the network monitoring system. For the GEN II Honeynet employed, monitoring occurred within the bridging firewall.

### **6.3 Data analysis**

The data that is collected on the Honeynet is stored in two separate locations on the monitoring system. Alerts that are triggered by the SNORT signature database are stored in an SQL database. These alerts are retrievable for display by using the ACID console. The Analysis Console for Intrusion Detection, or ACID, was developed by the Computer Emergency Response Team (CERT). If an alert for an exploit does not exist then the record of the exploit launched against the Honeynet will not be displayed on the ACID console. The following figure shows the SNORT alert output on the web-based ACID console.



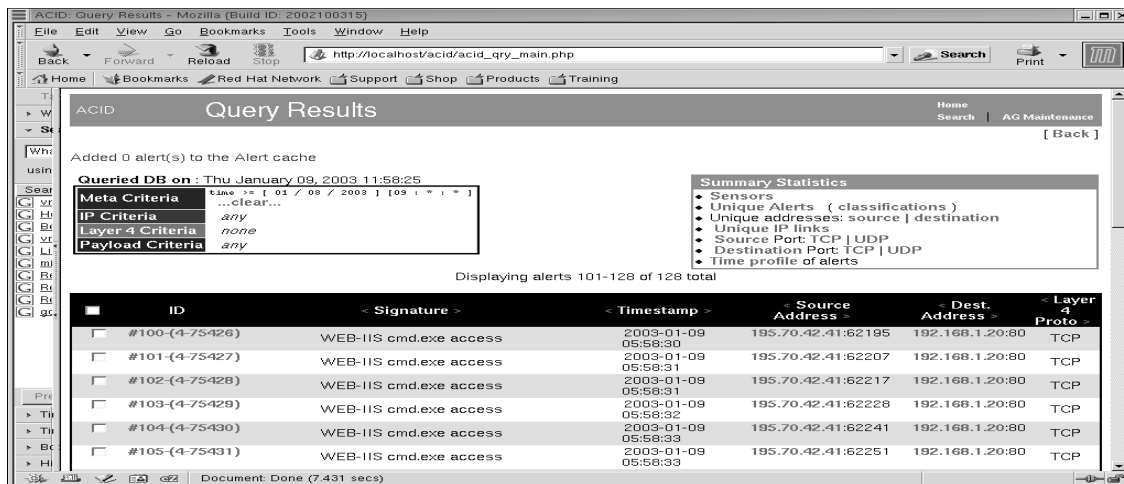


Figure 32: ACID Output of Honeynet Alerts

The additional data that is collected using the Data Capture capability of SNORT is stored in a daily log file on the monitoring system. A new directory is created each day for this data. We analyze this data using Ethereal. Ethereal is Open Source software that uses the libpcap library. Ethereal comes currently installed with most packaged installations of Linux software to include Red Hat and is available for download. Analyzing the data with Ethereal shows all of the traffic that was sent to or originated from the Honeynet. Ethereal displays the source and destination addresses of this traffic, protocol used, source and destination ports and packet content. It is also possible to collect all of the packet contents that correspond to a particular TCP session between an attacking machine and the Honeynet machine. It is possible to recognize automated worm type sweeps for various vulnerabilities as well as specific targeted attacks (both automated worm-type as well as manual) against specific services by analyzing the Honeynet data via Ethereal. A sample of the Ethereal output is displayed in the following figure (Figure 33).

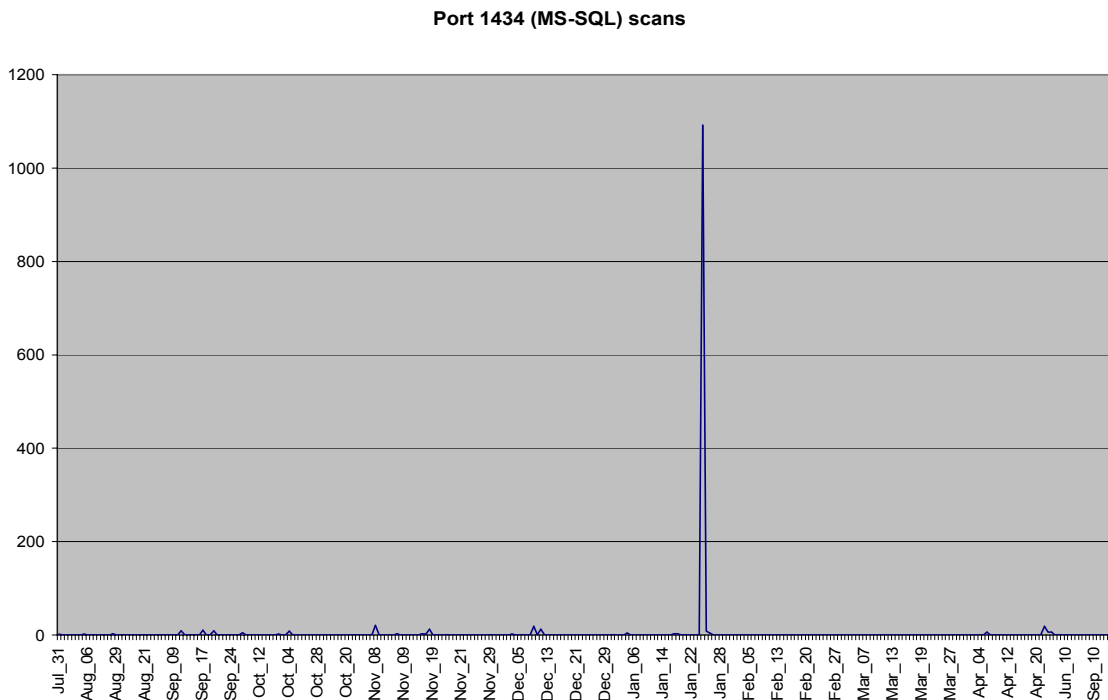


Figure 33: Ethereal screen shot of HoneyNet Data

The data provide by SNORT is analyzed on a daily basis. This analysis can be very time consuming. We spend at least one hour per day analyzing our HoneyNet data for our three computer network HoneyNet. When the HoneyNet is attacked and compromised by a hacker we usually spend much longer analyzing the data and conducting a forensic analysis on the compromised system or systems. One hour of attack traffic can equate to up to forty hours of analysis. When we finish analyzing the previous day's data, we can archive this data to a read-only medium. We are currently using the X-CD-Roast package on a LINUX machine to archive our data to a CD-ROM. X-CD-Roast is Open Source Software [52] which is in keeping with our goal of using this type of software. It has the ability to create multi-session copies on the same CD. This capability is necessary for us since on average we collect less than one megabyte of data per day. This data can be provided to other organizations for analysis.

## 6.4 Statistical Analysis of Honeynet Traffic

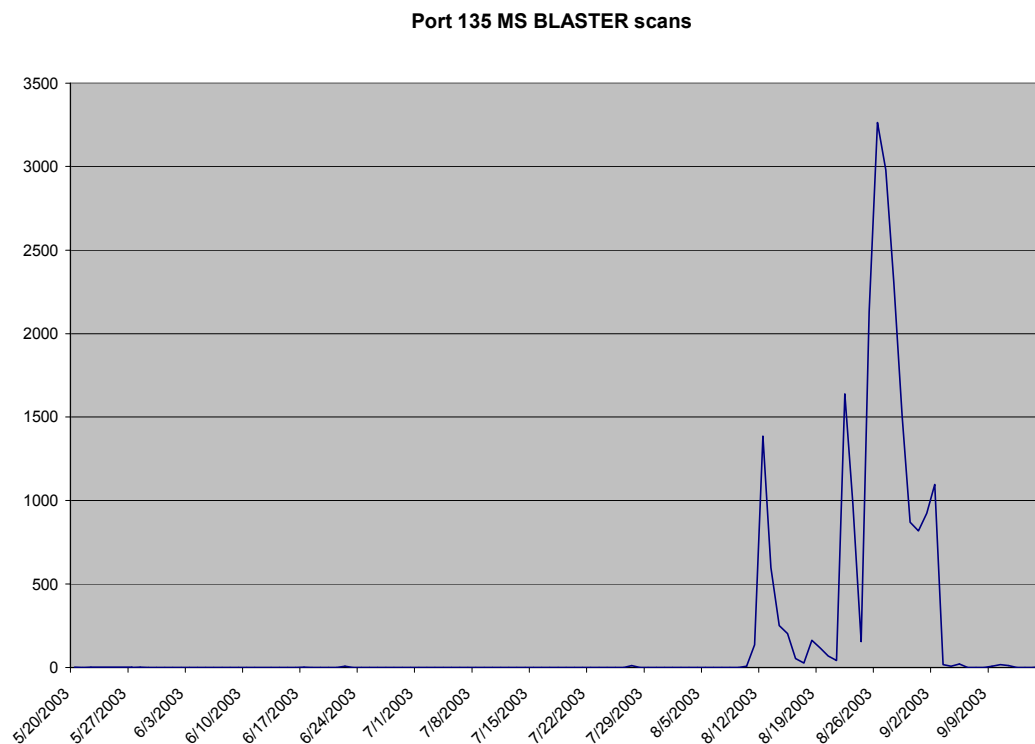
We have been able to conduct an analysis of various worm type exploits that have been launched against the Georgia Tech campus network. The first figure below shows the Microsoft SQL Slammer worm traffic directed against the Honeynet. This worm struck the Internet on 25 January 2003, which is the same day that the dramatic increase in traffic occurred on the Honeynet. There was some low level traffic directed against the port that this exploit targeted (TCP port 1434) prior to 25 January 2003. Microsoft published an exploit warning concerning this service (SQL) on 24 July 2002. It may have been possible to use the fact that low level traffic was being directed against this port as well as the fact that a published exploit existed against this port to anticipate that this port may have been subject to an exploit attack.



**Figure 34: SQL Slammer Worm**

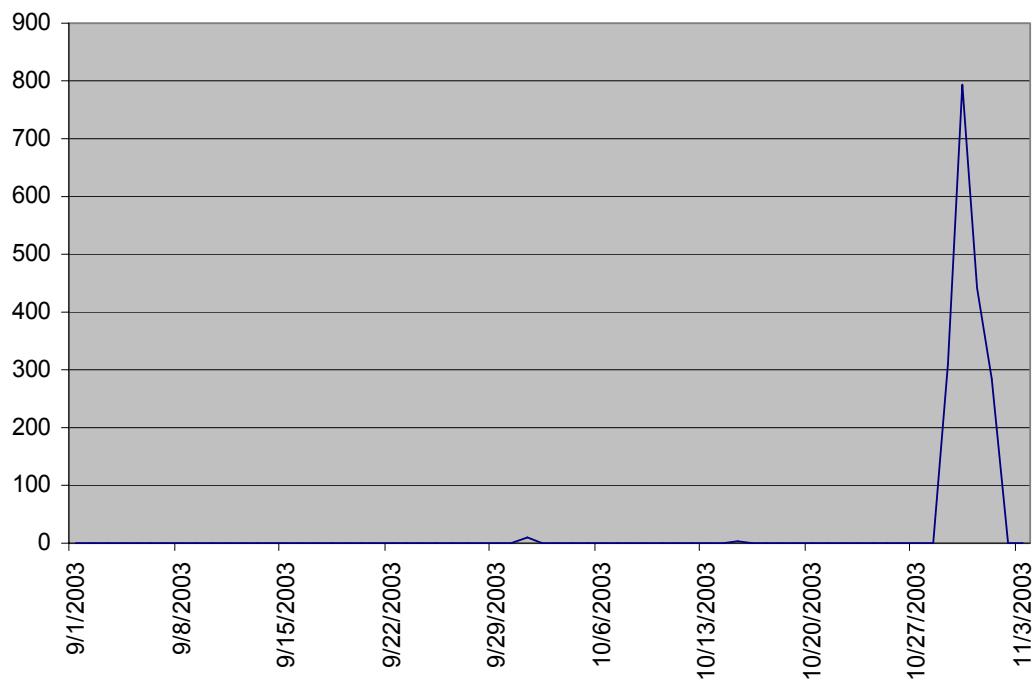
System administrators could have used the fact that an exploit existed against this particular port and that low but increasing levels of scans were occurring against this port to take countermeasures to secure the service running at this port. These countermeasures could include installing the most recent patches for the affected service or to block off-campus access to the target port.

The next figure below shows the recent exploit traffic directed against the Honeynet from the Microsoft Blaster worm. An exploit against this port was published by Microsoft on 16 July 2003 and this exploit was launched against the Internet on 11 August 2003. It is very significant to note that the time between exploit publication and worm attack has shrunk considerably from approximately six months for SQL Slammer to less than one month for the Blaster worm [53].



**Figure 35: Microsoft RPC (port 135) exploit**

Variations of the Blaster and the Nachi worm continue to appear on the Internet. One possible variant was detected by the Georgia Tech Honeynet on 26 October 2003. Prior to this date the level of Blaster and Nachi traffic was significantly reduced. On 29 October 2003 we were able to isolate the exploit that was being used to compromise the Microsoft 2000 system on the Honeynet via a Remote Procedure Call exploit. Analysis of this exploit and the files that were uploaded on the compromised machine result in a characterization of this exploit as the original Nachi exploit. The following figure shows the level of traffic involving this exploit that was being directed against the Georgia Tech Honeynet.



**Figure 36: Honeynet RPC Exploit**

This specific exploit targeted TCP port 707 on a system that was compromised by the RPC exploit. The drop in traffic occurring on 3 November 2003 was not due to a

reduction in overall traffic targeting port 707. It was a result of a hacker who compromised this specific system using a similar exploit. This hacker, upon compromising this system, disabled the RPC port on this system and then installed an Internet Relay Chat (IRC) bot (robot) on this system. Disabling the RPC port reduced the level of traffic targeting the vulnerable port on the Honeynet machine. Investigation of this IRC channel indicated that it was being used to illegally download copyrighted material. Further investigation revealed that 26 Georgia Tech machines belonged to this IRC channel. These Georgia Tech machines may have been compromised in a manner similar to the compromise that occurred on the Honeynet machine.

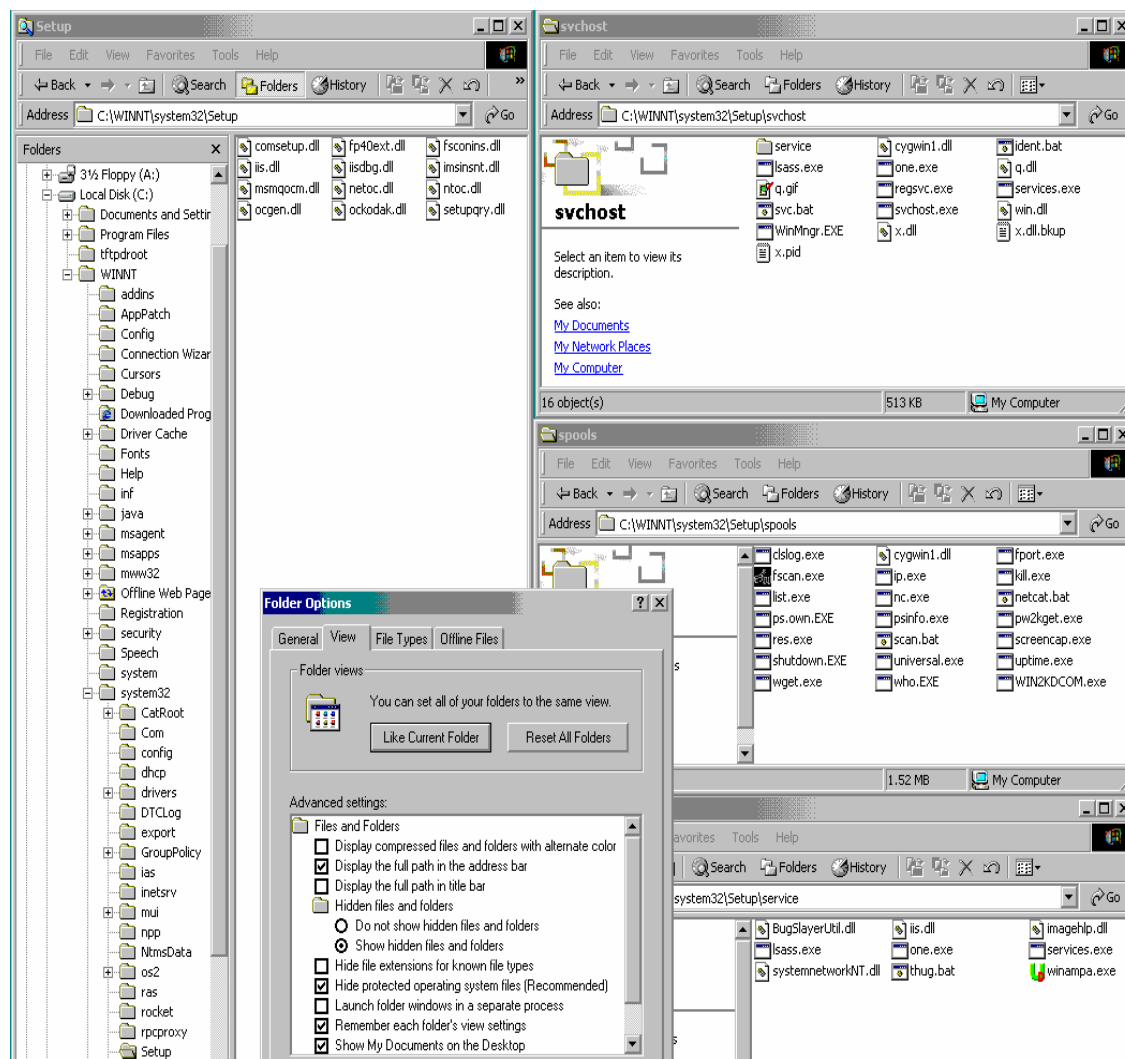
Although our research has been primarily focused on the Linux operating system, the Honeynet was able to also produce information concerning the exploit of other operating systems (in this case Microsoft 2000) that could be used to detect and classify system exploits. We conducted an analysis of the Microsoft 2000 Honeynet machine to try and recognize the specific delta ( $\nabla$ ) characteristic that could be used to identify the rootkit that may have been installed on this system.

Analysis of the Honeynet logs for this exploit showed the actions the hacker took upon gaining administrator privileges on this machine via the RPC exploit. The hacker used some form of the original Microsoft Blaster exploit to establish a backdoor Trojan connection with administrator privileges on tcp port 4444 on the compromised system [54]. The hacker was connecting to the Honeynet system from another system within the Georgia Tech domain (fgd06.eastnet.gatech.edu -128.61.100.134). We are unable to tell if the attack was originating from this address or if this machine was being used as a relay for the launch of the attack.

The hacker then connected to this system and conducted the following:

1. Created a batch file to conduct an ftp session with a remote computer (128.8.49.73 ) using a username of fbi and a password of cia
2. Downloaded two files named c.exe and x.exe
3. Deleted the batch file used for the ftp download.
4. Executed the programs c.exe and x.exe
5. Moved to the directory C:\WINNT\system32\Setup\svchost and starts the IRC bot with the batch file svc.bat. This directory and the file in it did not exist prior to the system compromise. We believe this directory and the files that it contains are a result of the c.exe or the x.exe program that was executed by the hacker.
6. The hacker then starts the services that he has associated with the warez server capability on this system.
7. A copy of the Microsoft Windows 2000 DCOM server program (WIN2KDCOM.exe) is downloaded from a machine at Old Dominion University (128.28.235.158 – garnet.test.odu.edu) and executed on the system.
8. The hacker then deletes the c.exe and x.exe programs that were downloaded. The hacker also hides the new directories that have been created under the C:/WINNT/system32/Setup. There are three directories created by the hacker:
  - C:/WINNT/system32/Setup/svchost
  - C:/WINNT/system32/Setup/spools
  - C:/WINNT/system32/Setup/service

These directories contain the various tools that the hacker uses to run the IRC bot and the warez server. Figure 37 shows these hidden directories.

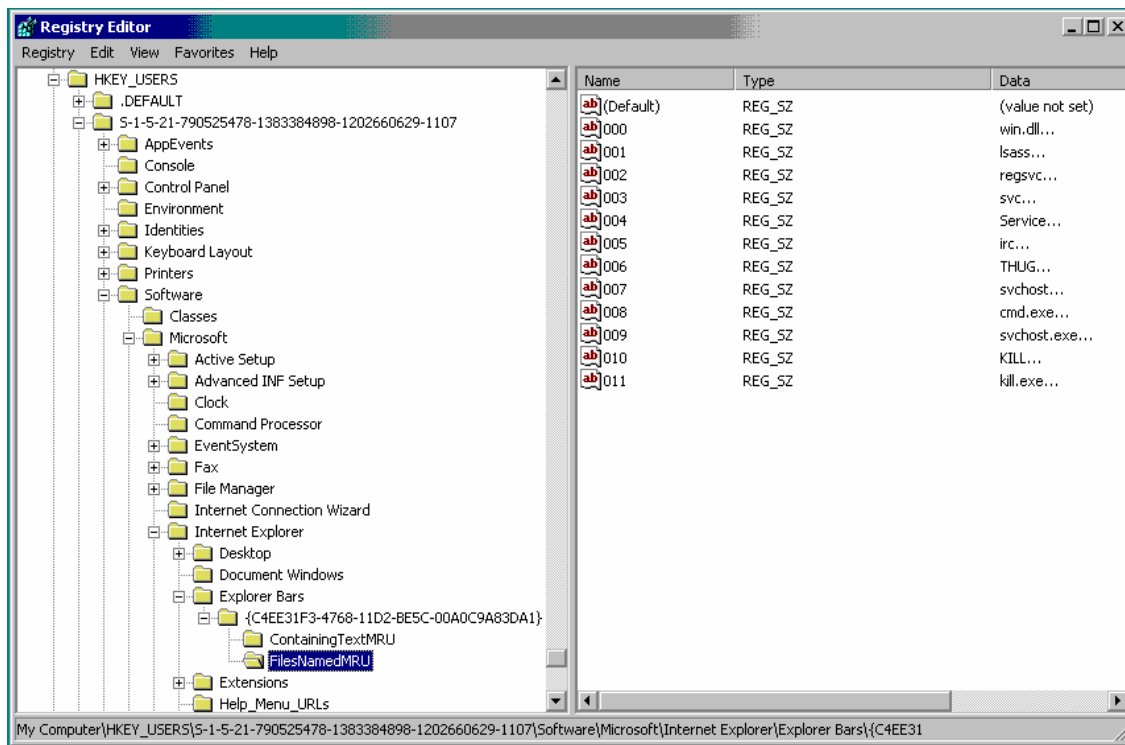


**Figure 37: Hidden directories created by hacker**

The figure shows that there are no subdirectories associated with the C:\WINNT\system32\setup directory that are displayed by the Microsoft NT Explorer tool even when this tool is set to display all hidden files and directories. However, as indicated by the other instances of NT Explorer in the figure, three subdirectories exist.



We were able to access these subdirectories by references their exact path as indicated in the Honeynet logs for this exploit. The Hacker also modified the Registry of this system to restart the IRC bot and the warez service upon system reboot. Figure 38 shows the modifications that were made to the Registry. The registry values listed are the files the hacker used when the machine was initially compromised.



**Figure 38: Modified Registry Entries on Compromised MS Machine**

Although our research has been primarily focused on the Linux operating system, we were able to apply elements of our methodology as well as data connected from the Honeynet to identify delta ( $\nabla$ ) characteristics of a Microsoft exploit. These  $\nabla$  characteristics consisted of three new directories being created on the target system as well as identifying changes made to the registry. This delta  $\nabla$  can then be used to conduct a classification of this Microsoft exploit.

## **6.5 Summary**

We have demonstrated in this chapter that a Honeynet can be used as a research tool to collect information about rootkits for follow-on analysis. This information can be used to classify rootkits as existing, modifications to existing, or entirely new. The Honeynet can also serve to increase the overall level of security on the network where the rootkit is installed. In the next chapter we present a case study concerning the analysis of a previously unseen Linux rootkit that was installed on a Honeynet system.

## **Chapter 7**

### **Detection and Analysis of a Previously Unseen Rootkit**

Part of our research, as stated in Chapter 6, is the use of a Honeynet to collect new rootkit type exploits. Rootkits are also available from other sources to include the Internet. The Honeynet, however, offers us an opportunity to collect rootkits that may not have been previously seen by other researchers [55]. These rootkits are targeted against actual live systems on the Honeynet that have been compromised by a hacker. We believe that the Honeynet offers us an actual opportunity to collect existing, modification to existing, and entirely new rootkit exploits.

On 1 June 2003 a system installed on the Georgia Tech Honeynet was compromised, allowing a hacker to gain root level access. The hacker then installed a rootkit on this system. No traffic should have been going to or come from this system since it is a honeynet machine. By following the principles of data capture and data control we were able to capture the exploit that the hacker executed against this system and prevent this system from being used to compromise any other systems.

#### **7.1 Target System Description**

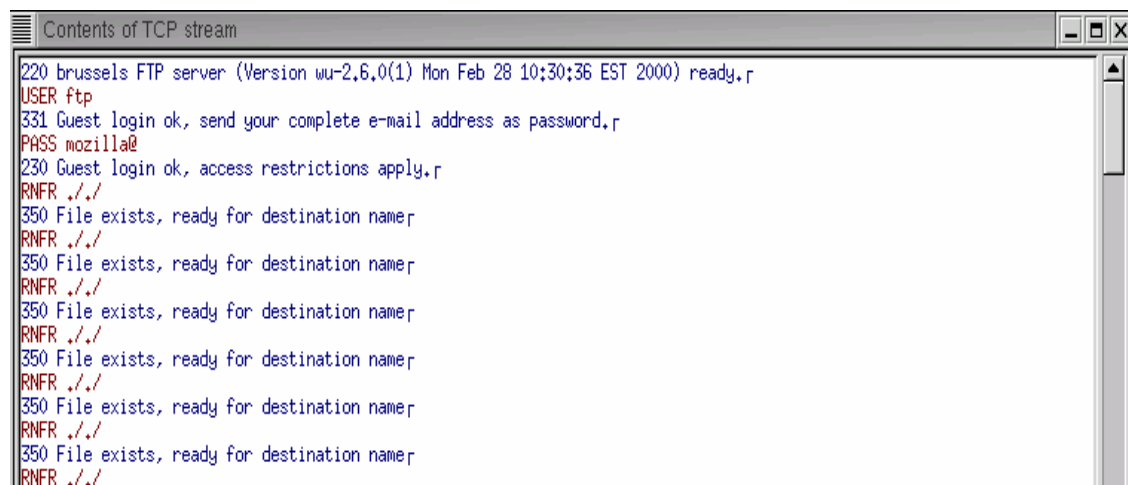
The target system that was employed on the Honeynet was a standard version of the Red Hat Linux 6.2 operating system running the Linux 2.2 kernel. This system was configured to install all available packages and no special modifications were made to this system. The install process was a default installation for this configuration. No additional services besides those that were started by the default installation were enabled

on the target system. The following ports were opened on this system:

21 ftp	98 Linux conf	514 shell	1025 listen
23 telnet	111 sun rpc	515 printer	1033 net info
25 smtp	113 auth	954 unknown	6000 x11
79 finger	513 login	1024 kdm	

## 7.2 Method of Compromise

At 06:34 on the morning of June 1, 2003 an exploit was launched against the target system on port 21 (ftp daemon) to attempt to gain root level access. Red Hat Linux 6.2 installs with the wu-ftpd2.6.0(1) ftp daemon. Exploits that allow a hacker to gain root level access have been published against this particular service and are available on the Internet. This attack was successful and the hacker was able to gain root level access on the target system. Figure 39 below shows the start of the tcp stream that was extracted from the Honeynet data concerning this attack. The string RNFR ././ is a signature of the WU-FTP exploit for the ftp server that is running on this system [56].



```
Contents of TCP stream
220 brussels FTP server (Version wu-2,6,0(1) Mon Feb 28 10:30:36 EST 2000) ready.r
USER ftp
331 Guest login ok, send your complete e-mail address as password.r
PASS mozilla@
230 Guest login ok, access restrictions apply.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
350 File exists, ready for destination name.r
RNFR ././
```

Figure 39: Start of Exploit



downloaded through a telnet session using the wget command. We do not believe that this particular rootkit has been analyzed before. There is an ssh rookit called 'r.tgz' but the characteristics of that rootkit, to include the file size, differ from the rootkit that was installed on the target system [57]. In any event, we were unable to find any detailed examination of a rootkit called 'r.tgz' with characteristics similar to the one that was installed on the target system.

The hacker extracts the exploit code within the 'r.tgz' file and then runs the exploit on the target system. Figure 41 shows the actual Honeynet data of the information that was provided back to the hacker. The 'r.tgz' rootkit deletes all traces of itself on the target system after installation. However, we are able to reconstruct what the hacker accomplished by utilizing the Honeynet logs for this exploit session.

```

Contents of TCP stream
setup
./setup
[1:36m Starting Instalton Of Rootkit ... [0m
[1:33m Step [1:35m: [1:37m1 ... [0m
[1:32m[+] [1:36mStarting Creating Our Local Directory And Moveing Programs ... [1:32m[+] [0m
[0:32m[--- [1:36m Moving & Copying Utils Programs On Our Directory ... [0:32m[---] [0m
[0:32m |--- [1:33mcuratare[0m
[0:32m | \[0m
[0:32m | |--- [1:36mdone moveing [1:33mcuratare[0m
[0:32m |--- [1:33mcl & X[0m
[0:32m | \[0m
[0:32m | |--- [1:36mdone copying [1:33mcl & X[0m
[0:32m |--- [1:33mfirewall status & socklist[0m
[0:32m | \[0m
[0:32m | |--- [1:36mdone copying [1:33mfirewall , status & socklist[0m
[0:32m |--- [1:33mread & write[0m
[0:32m | \[0m
[0:32m | |--- [1:36mdone copying [1:33mread & write[0m
[0:32m |----[0m
[1:33m Step [1:35m: [1:37m2 ... [0m
[1:32m[+] [1:36mStarting FireWall Rules ... [1:32m[+] [0m
[1:33m Step [1:35m: [1:37m3 ... [0m
[1:32m[+] [1:36mRemoveing Other Rootkits [1:33m( If They Exist Here) [1:36m... [1:32m[+] [0m
[1:33m Step [1:35m: [1:37m4 ... [0m
[1:32m[+] [1:36mReplaceng Some Files [1:33m( If They Exist Here) [1:36m... [1:32m[+] [0m
type port inode uid pid fd name
tcp 23 23141 0 13001 2 in.telnetd
Entire conversation (11689 bytes)
[ ] [x] ASCII [ ] EBCDIC [Print] [Save As] [Filter out this stream] [Close]

```

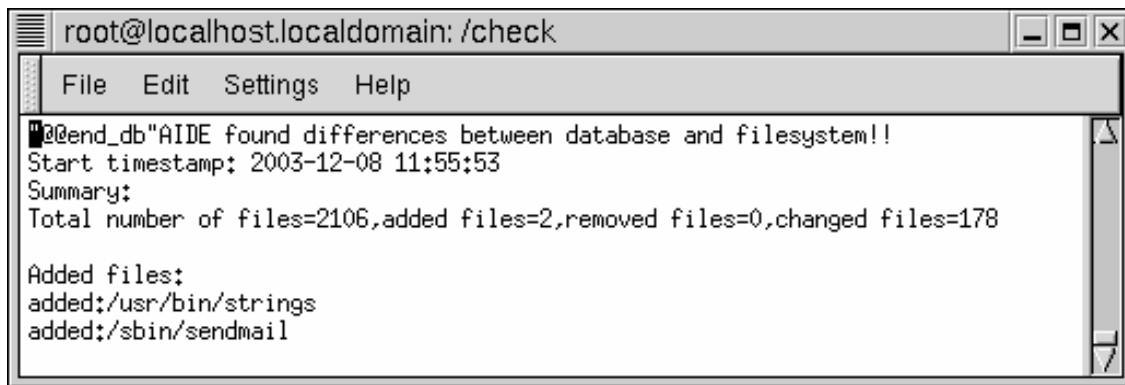
Figure 41: Installation of 'r.tgz' rookit

### 7.3 Analysis Process

The Georgia Tech Honeynet was able to capture the entire exploit session for this system compromise to include downloaded files as well as the remote machines that the hacker connected to from the compromised machine. This provided us with the scripts and files that were used by the hacker to install the r.tgz rootkit. Preliminary analysis of these files gave us an indication of how this rootkit would install on a target system similar to the system on the Honeynet. We then set out to install the r.tgz rootkit on a target system to analyze and classify it.

We have initially set up a baseline system that consists of the same operating system as the system that was compromised on the Honeynet. In this case it is Red Hat 6.2 running the Linux 2.2.14 kernel. Following the procedures outlined in Chapter 4, we installed a kernel level debugger (kdb) on this system as well as a file integrity checker program (AIDE) . We then installed a known rootkit detection program (chkrootkit) and made a copy of the kernel text segment via /dev/kmem. Prior to infecting this system with the r.tgz rootkit we ran the AIDE and chkrootkit program to establish a clean baseline for analysis and classification. We then infected this system with the r.tgz rootkit.

The first check that we ran on the infected system is the file integrity check to determine what files have been added, changed, or deleted. Running AIDE on the infected system indicates that 2 files have been added to the infected system and 178 files have been changed by the r.tgz program. This is a large number of files and initial analysis of the install scripts for this rootkit does not indicate that all of these files are modified. Follow-on analysis is conducted on these modified files to determine the nature of these changes. Figure 42 shows the results of running AIDE on the infected system.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the following text:

```
##end_db"AIDE found differences between database and filesystem!!  
Start timestamp: 2003-12-08 11:55:53  
Summary:  
Total number of files=2106,added files=2,removed files=0,changed files=178  
  
Added files:  
added:/usr/bin/strings  
added:/sbin/sendmail
```

**Figure 42: AIDE results on r.tgz infected system**

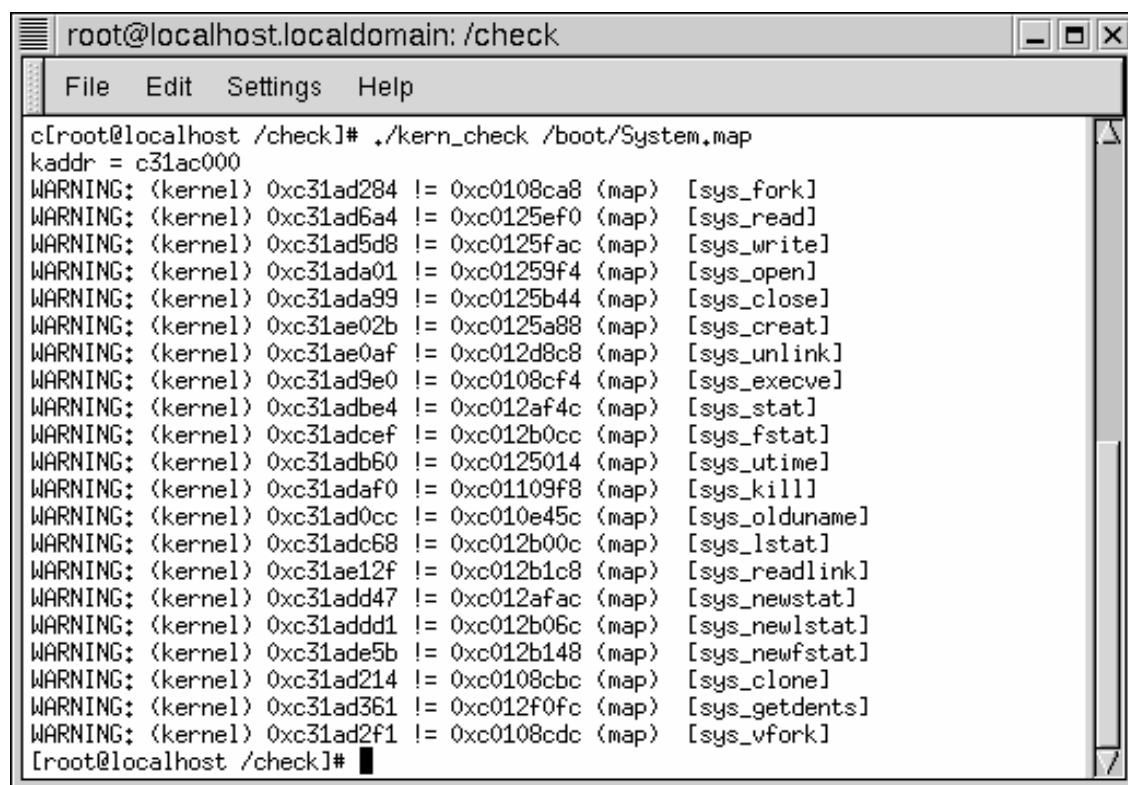
The next step was to run the Known Rootkit Detection Program (chkrootkit) on the target system. Running this program utilizing the binaries that are currently installed on the target system only results in the identification of one system binary as being infected. The binary that is indicated as being infected is *ifconfig*. The chkrootkit program also detects five suspicious files and possible infections by the Showtee and Romanian rootkits.

The recommended method of using the chkrootkit program is to use known good binary files. Using known good binary files results in the identification of the same changes identified in the previous paragraph plus 5 additional binaries being identified as being infected on the target system. These five binary files are: *du*, *ifconfig*, *killall*, *ls*, and *ps*. The check using known good binaries also indicates the following under the lkm check: 1 process hidden from readdir command, 15 processes hidden from ps command. The differences between the output using known good binaries and the output using the binaries currently installed on the system indicate that known good binaries should always be used while running the chkrootkit program. The five files detected as being changed by the chkrootkit program are also detected as being changed by the AIDE file



integrity check program. These results are used in our methodology to classify this rootkit exploit.

The next step in the methodology is to verify the integrity of the kernel. Running the kernel check program (`kern_check`) on the target system utilizing the `/boot/System.map` file indicates that there is a mismatch of 21 system calls between the kernel and the `/boot/System.map` file. The results of the `kern_check` program are shown in figure 43.



```
root@localhost.localdomain: /check
File Edit Settings Help
c[root@localhost /check]# ./kern_check /boot/System.map
kaddr = c31ac000
WARNING: (kernel) 0xc31ad284 != 0xc0108ca8 (map) [sys_fork]
WARNING: (kernel) 0xc31ad6a4 != 0xc0125ef0 (map) [sys_read]
WARNING: (kernel) 0xc31ad5d8 != 0xc0125fac (map) [sys_write]
WARNING: (kernel) 0xc31ada01 != 0xc01259f4 (map) [sys_open]
WARNING: (kernel) 0xc31ada99 != 0xc0125b44 (map) [sys_close]
WARNING: (kernel) 0xc31ae02b != 0xc0125a88 (map) [sys_creat]
WARNING: (kernel) 0xc31ae0af != 0xc012d8c8 (map) [sys_unlink]
WARNING: (kernel) 0xc31ad9e0 != 0xc0108cf4 (map) [sys_execve]
WARNING: (kernel) 0xc31adb4e != 0xc012af4c (map) [sys_stat]
WARNING: (kernel) 0xc31adcef != 0xc012b0cc (map) [sys_fstat]
WARNING: (kernel) 0xc31adb60 != 0xc0125014 (map) [sys_utime]
WARNING: (kernel) 0xc31adaf0 != 0xc01109f8 (map) [sys_kill]
WARNING: (kernel) 0xc31ad0cc != 0xc010e45c (map) [sys_olduname]
WARNING: (kernel) 0xc31adc68 != 0xc012b00c (map) [sys_lstat]
WARNING: (kernel) 0xc31ae12f != 0xc012b1c8 (map) [sys_readlink]
WARNING: (kernel) 0xc31add47 != 0xc012afac (map) [sys_newstat]
WARNING: (kernel) 0xc31addd1 != 0xc012b06c (map) [sys_newlstat]
WARNING: (kernel) 0xc31ade5b != 0xc012b148 (map) [sys_newfstat]
WARNING: (kernel) 0xc31ad214 != 0xc0108cbc (map) [sys_clone]
WARNING: (kernel) 0xc31ad361 != 0xc012f0fc (map) [sys_getdents]
WARNING: (kernel) 0xc31ad2f1 != 0xc0108cdc (map) [sys_vfork]
[root@localhost /check]#
```

**Figure 43: Results of `kern_check` program**

This is an indication that the kernel of the target system may have been compromised by the `r.tgz` rootkit. Checking the current kernel text segment code against the previously archived version of the kernel text segment code that was built when the target system was first compiled prior to infection also indicates that the kernel has been compromised.

Analysis of the kernel using kdb indicates that the pointer to the system call table is being redirected to a new instance of this table. The correct system call table address is 0xc0248928 and can be retrieved from the /boot/System.map file. The current system call table address as displayed by kdb in kernel memory is 0xc31ac000. The following is the results of running a kdb query on the system call interrupt within kernel space. The returned call statement should refer to the address of the system call statement that is stored in the /boot/System.map file (0xc024928) and it does not.

```
kdb> id 0xc0109d84 ~ (address of system call interrupt from /boot/System.map)
system_call + 0x2d:      call *0xc31ac000(,%eax,4)
```

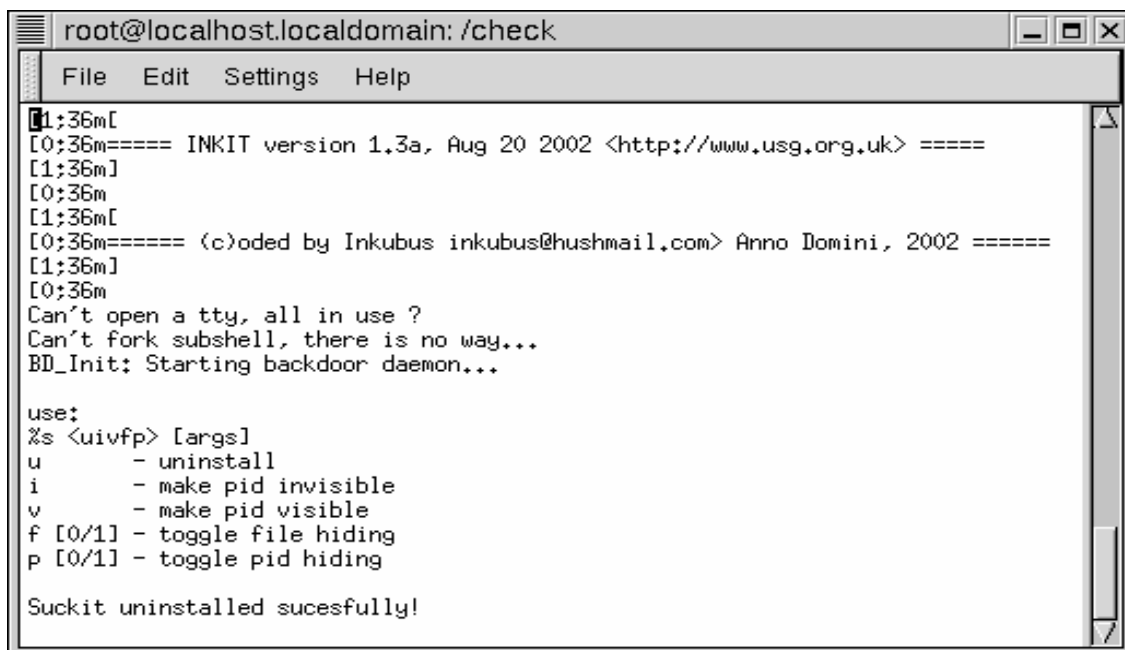
This is an indication that the system call table is being redirected by a kernel rootkit

Since the kernel is a key part of the computer operating system we will first examine this aspect of the r.tgz rootkit to determine the method that this rootkit used to compromise the target system. You can not trust any of the system output if the kernel has been compromised.

As previously mentioned, the Honeynet allowed us to retrieve the install scripts and code that is utilized by the r.tgz rootkit. The main install file for the r.tgz rootkit calls a series of script files to install the rootkit. Analysis of these scripts indicates that the startfile script is the script that compromises the kernel on the target system. The replace script is used to replace the system binaries.

The startfile script copies the r.tgz file *init* to the /etc/rc.d/init.d directory. Analysis of the r.tgz *init* file indicated that this is the script that compromises the kernel. The *init* file executes several binary files named *sendmail* (executed as a daemon), *write* (executed as a daemon), and two instances of the executable *all* with the 'i' switch and a pid. The

*sendmail* binary file is actually another instance of the *all* program that is copied over in the *createdir* script file that is executed by the *r.tgz* startup script file. This analysis resulted in the identification of three instances of the *all* binary executable file being executed by the *r.tgz* rootkit. Two of these instances of the *all* binary file have pid's associated with them. The fact that the *init* file, which calls these three instances of the *all* file, has been copied into the */etc/rc.f/init.d* directory is an indication that the *r.tgz* rootkit developer wanted this code to be executed upon system reboot, making the kernel compromise portion of this rootkit resident within memory. As a result of this analysis, we choose to examine the *all* program. This program is a binary executable file and we do not have the underlying source code that was used to create this rootkit. We choose to use a tool such as *strings* on this file initially in conducting our analysis. An segment of the results of running the *strings* command on the *r.tgz all* binary file are indicated in Figure 44.



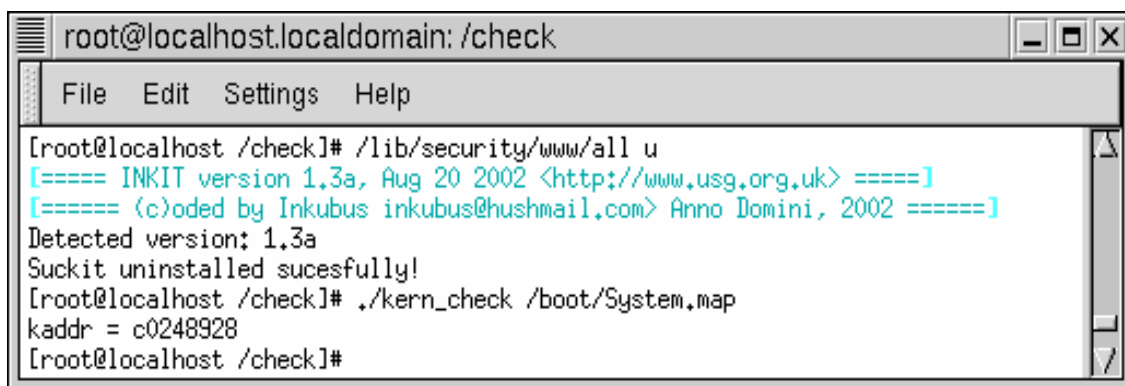
```
root@localhost.localdomain: /check
File Edit Settings Help
[1:36m[
[0:36m===== INKIT version 1.3a, Aug 20 2002 <http://www.usg.org.uk> =====
[1:36m[
[0:36m[
[1:36m[
[0:36m===== (c)oded by Inkubus inkubus@hushmail.com> Anno Domini, 2002 =====
[1:36m[
[0:36m[
Can't open a tty, all in use ?
Can't fork subshell, there is no way...
BD_Init: Starting backdoor daemon...

use:
%s <uivfp> [args]
u      - uninstall
i      - make pid invisible
v      - make pid visible
f [0/1] - toggle file hiding
p [0/1] - toggle pid hiding

Suckit uninstalled sucesfully!
```

**Figure 44:** strings output of *r.tgz all* program

The strings output of this program indicates that the *all* program is a kernel level rootkits known as INKIT. A search on the Internet for a kernel rootkit called INKIT does not result in any references to this particular rootkit. According to the use statement that is output by the *strings* command the 'i' switch that is used in the *init* script with a particular pid is used to make that pid invisible. The last string displayed in the figure is significant to note. This text string makes reference to the SuckIT rootkit. We present an in depth analysis of the SuckIT kernel level rootkit in the appendix. The fact that this string appears in INKIT, including the misspelling of successfully just like the actual SuckIT rootkit, leads us to believe that INKIT is a modification or a copy of the SuckIT kernel level rootkit. We then attempt to uninstall the INKIT kernel level rootkit using the 'u' switch as indicated by the use statement in Figure 45.

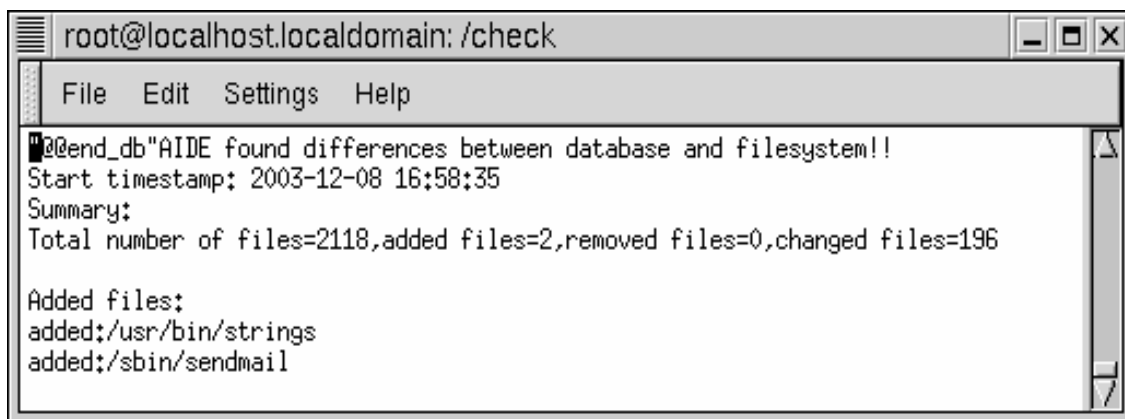
A terminal window titled 'root@localhost.localdomain: /check' with a menu bar (File, Edit, Settings, Help). The terminal shows the execution of the command '/lib/security/www/all u'. The output includes a cyan-colored header: '==== INKIT version 1.3a, Aug 20 2002 <http://www.usg.org.uk> =====' and '(c)oded by Inkubus inkubus@hushmail.com> Anno Domini, 2002 ====='. Below this, it says 'Detected version: 1.3a' and 'Suckit uninstalled sucesfully!'. The next command is './kern\_check /boot/System.map', which outputs 'kaddr = c0248928'. The prompt returns to 'root@localhost /check#'.

```
root@localhost.localdomain: /check
File Edit Settings Help
[root@localhost /check]# /lib/security/www/all u
[==== INKIT version 1.3a, Aug 20 2002 <http://www.usg.org.uk> =====]
[==== (c)oded by Inkubus inkubus@hushmail.com> Anno Domini, 2002 =====]
Detected version: 1.3a
Suckit uninstalled sucesfully!
[root@localhost /check]# ./kern_check /boot/System.map
kaddr = c0248928
[root@localhost /check]#
```

**Figure 45: Uninstall of INKIT kernel rootkit**

This command is successful in uninstalling the kernel level rootkit. We then verify the integrity of the kernel with the *kern\_check* kernel integrity check program. The indication is that there are no system calls currently being redirected by the kernel. An examination of the kernel using *kdb* indicates that the system call interrupt is now referencing the correct system call table. As a final check, we compare the current kernel

text segment against the original archived text segment. These files now match. At this point the kernel is no longer compromised and we rerun the file integrity checker and known rootkit detection programs on the target system since we can now trust the kernel output. The output from the AIDE file integrity check program now indicates that 196 files have changed. This is an increase of 19 files from the previous check of the AIDE program where 177 files were detected as being changed by the r.tgz rootkit. It appears that the kernel element of the r.tgz rootkit was hiding these 19 changed programs from the AIDE program. We will analyze these changed files and the r.tgz install scripts to determine how these files have been changed. The results of the new instance of the AIDE program are indicated in Figure 46.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the AIDE program's results: '@@end\_db"AIDE found differences between database and filesystem!!', 'Start timestamp: 2003-12-08 16:58:35', 'Summary:', 'Total number of files=2118,added files=2,removed files=0,changed files=196', 'Added files:', 'added:/usr/bin/strings', and 'added:/sbin/sendmail'.

```
root@localhost.localdomain: /check
File Edit Settings Help
@@end_db"AIDE found differences between database and filesystem!!
Start timestamp: 2003-12-08 16:58:35
Summary:
Total number of files=2118,added files=2,removed files=0,changed files=196
Added files:
added:/usr/bin/strings
added:/sbin/sendmail
```

**Figure 46: New AIDE results on target system.**

Analysis of the install scripts for the r.tgz rootkit does not indicate that 196 files are being changed when the rootkit is installed. Analysis of the AIDE results indicates that all of the executable files in the /bin directory are changing. Comparison of the files in the /bin directory with known good files indicates that the files that are not being changed by the r.tgz rootkit are increasing in size by 8759 bytes. This increase in file size is a

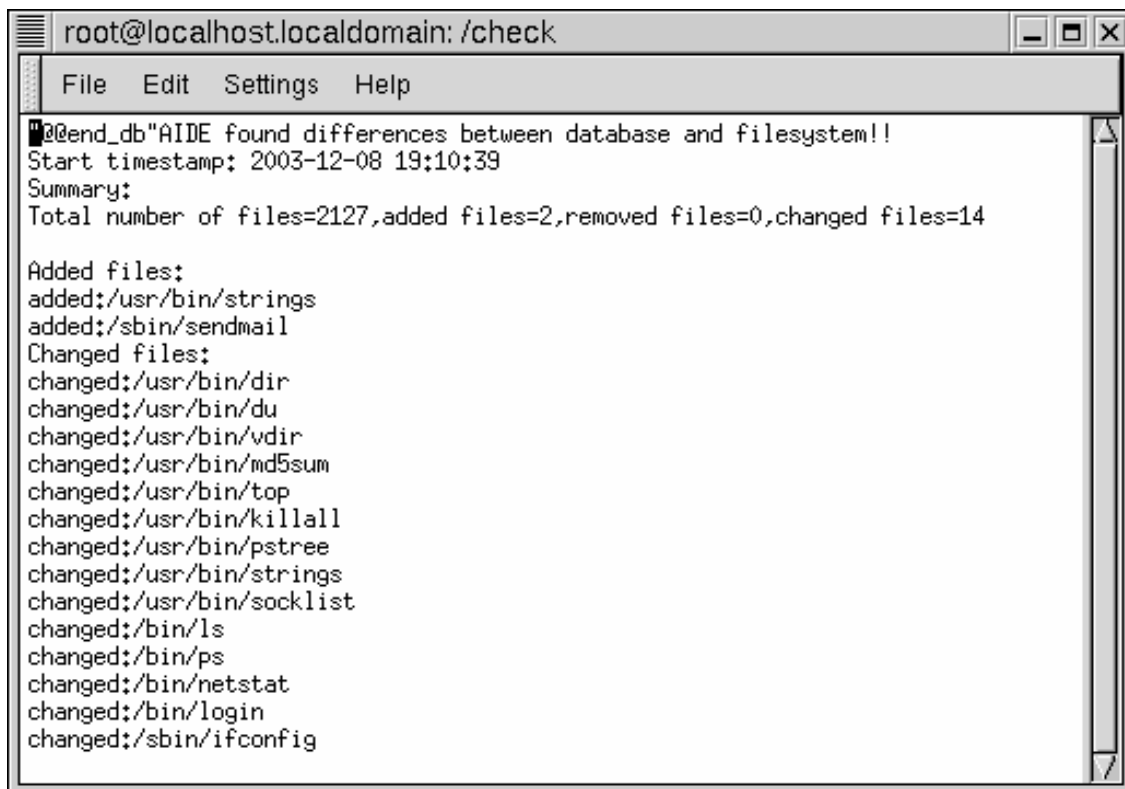
signature of the Linux.OSF.8759 virus. This virus is associated with the hax.tgz rootkit [58]. Application of the methodology thus far has indicated that r.tgz is composed of elements of two rootkits; the INKIT kernel level rootkit which is based on SuckIT, and the hax.tgz binary level rootkit.

Another signature of the Linux.OSF.8759 virus is that a trojan port is opened on the target system at 3049. This trojan port is detected by the chkrootkit program which checks for processes listening on ports with the use of the netstat command with the `-anp` switch to detect open ports on the system in question. Other Trojan ports can be detected in a similar fashion.

There is a utility named clean.OSF.8759-ps that can be used to clean infection of the Linux.OSF.8759 virus [59]. However, these files can not be cleaned by the root user after the r.tgz rootkit is installed on the target system. This is a result of the *socklist* script within the r.tgz rootkit changing the attributes on all of the files in the /bin directory with the *chattr* `+ASacdisu` command. Attributes on selected files in the /sbin and /usr/bin directories are also reset by the *replace* script within the r.tgz rootkit after these files are infected with the Linux.OSF.8759 virus. Files and directories with their attributes reset could be used as an indication that a possible rootkit is installed on the target system. These attributes must be turned off using the *chattr* `-ASacdisu` command to disinfect these files.

To produce an accurate count of the number of binary files that are added, deleted, or changed by the r.tgz rootkit the attributes of the files within the /bin, /sbin, and /user/bin directories must first be reset. Then these directories can be disinfected with the clean.OSF.9759-ps utility. Once these files and directories have been disinfected the

AIDE program is run to provide an accurate count of the files that have been changed on the target system by the r.tgz rootkit. Figure 47 shows the results of this instance of the AIDE program on the target system. The results are that 2 files have been added and 14 files have been changed by r.tgz. These results correspond with the install scripts for the r.tgz rootkit.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the AIDE program's results: 'AIDE found differences between database and filesystem!!', 'Start timestamp: 2003-12-08 19:10:39', 'Summary:', 'Total number of files=2127,added files=2,removed files=0,changed files=14'. It then lists 'Added files:' as '/usr/bin/strings' and '/sbin/sendmail', and 'Changed files:' as a list of 14 files: '/usr/bin/dir', '/usr/bin/du', '/usr/bin/vdir', '/usr/bin/md5sum', '/usr/bin/top', '/usr/bin/killall', '/usr/bin/pstree', '/usr/bin/strings', '/usr/bin/socklist', '/bin/ls', '/bin/ps', '/bin/netstat', '/bin/login', and '/sbin/ifconfig'.

```
root@localhost.localdomain: /check
File Edit Settings Help
@@end_db"AIDE found differences between database and filesystem!!
Start timestamp: 2003-12-08 19:10:39
Summary:
Total number of files=2127,added files=2,removed files=0,changed files=14

Added files:
added:/usr/bin/strings
added:/sbin/sendmail
Changed files:
changed:/usr/bin/dir
changed:/usr/bin/du
changed:/usr/bin/vdir
changed:/usr/bin/md5sum
changed:/usr/bin/top
changed:/usr/bin/killall
changed:/usr/bin/pstree
changed:/usr/bin/strings
changed:/usr/bin/socklist
changed:/bin/ls
changed:/bin/ps
changed:/bin/netstat
changed:/bin/login
changed:/sbin/ifconfig
```

Figure 47: Accurate AIDE count of changed files

## 7.4 Rootkit Characteristics

The chkrootkit program only detects 5 of the 14 files that are detected as being changed by the AIDE program. It may be possible to develop new signatures for the 9 changed files that are not detected by chkrootkit. These files are: *dir*, *vdir*, *md5sum*, *top*, and *strings* in the */usr/bin* directory; *ps*, *netstat*, and *login* in the */bin* directory; and *ifconfig* in

the /sbin directory. Two other possible signatures are the two added files, /usr/bin/strings and /sbin/sendmail. There are currently two instances of the *strings* command in the /usr/bin directory. The *socklist* script of the r.tgz rootkit copies the original /usr/bin/strings file to the following file: /usr/bin/strings<blank space>. This would make it hard to detect that this file has been tampered with under a visual examination. However, this is a signature that can be used to detect the presence of either the r.tgz rootkit or the binary elements that make up the r.tgz rootkit on the target system.

Using the elements of the methodology that we have proposed in section 5.1 we are able to detect some possible unique string signatures in the binary files that are replaced by the r.tgz rootkit. The following are potential string signatures that can be used by a program such as chkrootkit to detect the presence of the r.tgz rootkit binary file replacements.

/usr/bin/dir	“stpcpy”
/usr/bin/vdir	“/usr/include/file.h”
/usr/bin/md5sum	“/usr/local/share/locale”
/usr/bin/top	“proc_hackinit”
/usr/bin/strings	“/bin/su –“
/usr/bin/socklist	“bin/egrep –v”
/bin/ps	“/tmp/extfsRNV23Z”
/bin/netstat	“__bzero”
/bin/login	“cococola”

We have already characterized the kernel rootkit that is an element of r.tgz. This kernel rootkit is one that redirects the system call table to an entirely new system call table as described in section 2.4.2. Based on other analysis that we have done, we were able to



uninstall and reinstall this rootkit on the target system. It is significant to note that every new reinstallation of this kernel level rootkit will results in a new address in kernel space for the new instance of the compromised system call table.

As previously stated, we concluded that the r.tgz rootkit is a blended rootkit that contains elements of the INKIT kernel rootkit and the hax.tgz binary rootkit. The INKIT rootkit is based on SuckIT. The hax.tgz rootkit is based on bigwar.tgz rootkit [58].

## **7.5 Summary**

We have applied our methodology to a rootkit that we were able to acquire on the HoneyNet research network that we established at Georgia Tech. Our methodology enabled us to identify the binary elements that this rootkit replaced on the target system. New signatures were identified that can help to detect the presence of this rootkit.

The methodology enabled us to characterize the kernel element of this rootkit as a modification to an already existing kernel rootkit. We were able to uninstall this kernel rootkit on the target system.

The analysis that we have presented concerning the application of our methodology to a rootkit would benefit network administrators, researchers, and network security personnel in characterizing rootkits as well as provide methods to detect these rootkits.

## **Chapter 8**

### **Conclusions and Further Recommendations**

The objective of our research was to develop a methodology for detecting and classifying rootkit exploits. There is no standardized methodology at present to characterize rootkits that compromise the security of computer systems. The goal of our methodology was to provide network system administrators, network security personnel, and security researchers with techniques to determine if a rootkit has been installed on their systems and to be able to classify these rootkits as existing, modification to existing, or entirely new. Current methods may only indicate that a system may be infected with a rootkit without characterizing the rootkit. The ability to characterize rootkits will provide system administrators, researchers, and security personnel with the information necessary to take the best possible recovery actions. This may also help to detect and fingerprint additional instances and prevent further security instances involving rootkits. The vulnerabilities that exist in modern operating systems as well the proliferation of exploits that allow hackers to gain root access on networked computer systems provide the ability to install rootkits. System administrators need to be aware of the threats that their computers face from rootkits as well as the ability to recognize if a particular rootkit has been installed on their computer system. In this research, we focused on defining a mathematical framework to define rootkit exploits as well as a formalized methodology to follow to characterize rootkits. In this chapter, we summarize our methodology.

## 8.1 Contributions

- We conducted an in-depth analysis of rootkits to include a description of what constitutes a rootkit. Modern hackers use rootkits as a tool to conceal their activity on a compromised computer. Different levels of rootkits were examined to include application level rootkits, trojan or system utility rootkits and kernel rootkits. Application rootkits are programs installed by a hacker to allow for backdoor connectivity on the target system. They normally do not replace or modify any existing programs. Trojan utility rootkits will alter or replace some trusted binary utility on the target system. These utilities are normally used by system administrators to provide accurate information on the system in question. The two currently existing types of kernel level rootkits were described. These include kernel level rootkits that modify the system call table and kernel level rootkits that redirect the system call table. Both of these types of kernel rootkits do not install or replace any programs on the target system. Kernel rootkits are very difficult to detect and characterize using existing methods. We believe that it is necessary to have an in-depth understanding of rootkits to be able to detect and characterize them.
- We examined current rootkit detection and prevention methodologies. This included a review of current general public license (GPL) tools available on the Internet. Our research addressed strengths and limitations of these tools when used to both detect and classify rootkits. GPL programs that are designed to maintain system integrity and prevent rootkit exploitation were also examined. A significant issue concerning these tools is that they must already be installed on a target system prior to the attempted installation of a rootkit. These tools are to be considered “White Hat” kernel level

rootkits since they modify the kernel on the target system to protect this system from malicious rootkits. These tools are designed to maintain system integrity and have no capability to classify rootkits. We also examined “Black Box” or Dynamic Program Analysis as a method of detecting and classifying rootkit exploits. This method of analysis showed promise in detecting application or binary programs but we demonstrated a failure concerning the detection of a kernel rootkit. The reason for this is due to the fact that a kernel rootkit modifies the kernel, or “Black Box” that is to be analyzed. We then examined the current rootkit detection methodology that is employed by the campus network security personnel at the Georgia Institute of Technology. These methods currently include both manual and automated procedures. Our research indicated that there is no formalized methodology currently existing to detect or characterize rootkit exploits.

- We proposed a formal methodology to characterize rootkits. This methodology helps network system administrators in increasing the security level of their networks. It allows for characterization of threats that a network may face which in turn allow for the appropriate steps to be taken to best secure a network. This formal methodology is based on the identification of a specific delta ( $\nabla$ ), or identifiable characteristic between the rootkit being examined and the underlying program that is being used as a baseline for comparison. New signatures can be generated for rootkits based on the application of our methodology and the identification of these  $\nabla$  characteristics. We present a framework that can be used to define rootkits as existing, modification to existing, or entirely new. This provides for a formal method to classify rootkits. We then developed a methodology to include the

establishment of a testing platform and the sequential steps that are to be taken to identify characteristics to be used for rootkit detection and classification.

- We implemented methods to detect and classify rootkit exploits by identification of particular  $\nabla$  characteristics. These methods demonstrated the ability to identify  $\nabla$  characteristics in rootkits that replaced system utilities as well as those that target the kernel of the operating system. We proposed techniques that could be used to detect unique string signatures in binary rootkits as well as the ability to detect unique string signatures in binary rootkits that employ string hiding techniques. We also proposed a method to detect kernel rootkits that redirect the system call table. Application of these methods should result in a  $\nabla$  that can be classified as belonging to an existing rootkit, modification to an existing rootkit, or to an entirely new rootkit.
- We established a Honeynet to collect rootkits for our research. Our methodology requires that we possess a copy of the rootkit in order to follow our methodology to detect and classify it. In addition, we demonstrated that a Honeynet can also be used to help secure a large enterprise network such as the one at Georgia Tech. We conducted statistical analysis of various worm type exploits that were run against the Honeynet. Our research is focused on the Linux operating system but data collected from the Honeynet enabled us to apply elements of our methodology against another operating system (Microsoft 2000) that was exploited. We detected specific  $\nabla$  characteristics of this exploit that can be used to detect and classify it.
- We applied our methodology against a new rootkit that we retrieved from a compromised Linux system that was installed on the Honeynet. This rootkit was unknown to us and we could find no reference to a rootkit of this type on the Internet.

Our analysis revealed this was a ‘blended’ rootkit that contained elements of both binary and kernel rootkits. Application of our methodology enable us to identify specific  $\nabla$  characteristics that can be used to detect and classify this rootkit.

## 8.2 Future Research

Suggestions for future research:

- Establish a methodology to re-establish trust on a system that has been compromised by a rootkit. It should be possible to accomplish this as a result of the characterization of the rootkit. However, in order to accomplish this, there must be an underlying element within the operating system that can be trusted.
- Investigate the possibility of the distribution of rootkit characterization signature databases. Organizations such as the HoneyNet Alliance [83] exist to collaborate concerning the collection and analysis of HoneyNet data. Our methodology could be applied to the rootkits that are collected by the HoneyNet Alliance in order to build a repository of various rootkit characteristics.
- Automate the methodology of detecting and classifying rootkit exploits. At present, many of the processes within the methodology are manual. It may be possible to automate many of these procedures to target specific system implementations.

## **APPENDIX**

### **A.1 The Linux Rootkit IV (lrk4)**

The Linux RootKit IV (lrk4) was released in November of 1998 by Lord Somer. It includes the usual rootkit components: a sniffer, utilities to edit and erase log files, and Trojan replacement system utility programs [60]. More recent versions of the lrk rootkit exist. The source code for Version 5 is also available on the Internet in addition to lrk4 source code for systems with and without shadow passwords. There is also a precompiled version of lrk4 that is available for downloading [61]. The lrk4 code continues to be modified and improved upon. An update to lrk4 was posted on the internet as recently as 11 May 2000 [62].

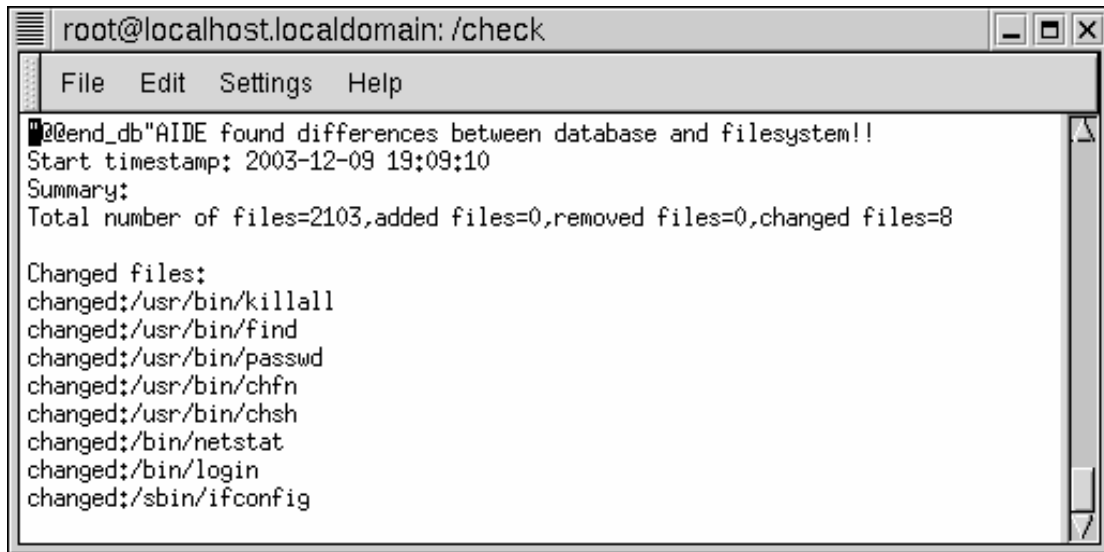
We followed our methodology in conducting an analysis of this rootkit. The following steps were taken prior to infecting a system with this rootkit:

1. The file integrity checker program AIDE was run on this system
2. A copy of the kernel text code was produced.
3. The known rootkit detection program, chkrootkit, was run on this system.

All of these steps were taken prior to infecting this system with the lrk4 rootkit. At this point the lrk4 rootkit was installed on the target system. The shadow password installation was utilized during this installation.

The AIDE program detected that 8 files had been changed on this system after the installation of the lrk4 rootkit. However, the chkrootkit program only detected that 6 files had been infected. The two files that were not detected by the chkrootkit program were

the *netstat* file and the *chsh* file. Figure 49 shows the changed files that were detected by the AIDE program.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the following text:

```
@@end_db"AIDE found differences between database and filesystem!!  
Start timestamp: 2003-12-09 19:09:10  
Summary:  
Total number of files=2103,added files=0,removed files=0,changed files=8  
  
Changed files:  
changed:/usr/bin/killall  
changed:/usr/bin/find  
changed:/usr/bin/passwd  
changed:/usr/bin/chfn  
changed:/usr/bin/chsh  
changed:/bin/netstat  
changed:/bin/login  
changed:/sbin/ifconfig
```

**Figure 48: AIDE results on lrk4 infected system**

The integrity of the kernel was not compromised by the installation of the lrk4 rootkit. The *kern\_check* program did not detect the redirection of any system calls. A comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched.

We then examined the two files that were not detected by the *chkrootkit* program to identify signatures that can be utilized for detection. Using the previously developed methods we identified the following potential signatures:

/usr/bin/chsh	“login.defs”
/bin/netstat	“written”

Based on this analysis we determine that the lrk4 rootkit is a Trojan Utility rootkit as described in section 2.3.



## **Analysis of the lrk4 login source code**

We chose to do a detailed examination of the login code that exists in the lrk4 rootkit written by Lord Somer. The login executable file was detected by both AIDE and the chkrootkit program. The AIDE program indicated the checksums for this file stored in the database no longer matched the current checksums produced for this file. The chkrootkit program indicated that this file was now infected.

The login code used by Red Hat 6.2 is derived from 4.3 BSD software, as indicated by the data provided from the Red Hat Package Manager (rpm) source code file [63]. The login code utilized in the lrk4 rootkit exploit is based on code written by John F. Haugh II as indicated by the code provided in the exploit [62]. The first significant difference between these two programs is that the clean version of the login program does not support the use of shadow passwords unless Pluggable Authentication Module (PAM) is available on the system. PAM is available on the Red Hat 6.2 installation of Linux. The hacked login code used in lrk4 uses the Shadow-Suite of software to support the use of shadow passwords.

The Shadow-Suite software is a package of shadow password file utilities. This package contains the necessary programs to convert traditional System V & UNIX password files to SVR4 shadow passwords. This package also contained the necessary programs to maintain the shadow and group files that work with both shadow and non-shadow passwords [64]. According to the man page, PAM is described as a system of libraries that handles the authentication tasks of applications and services on the system. Authentication is dynamically configurable with PAM.

Red Hat 6.2 did not install with the Shadow-Suite of software, but it did install with

PAM. The use of shadow passwords is established during the installation of the Red Hat 6.2 software, and PAM supports these shadow passwords. There is an rpm available to install the Shadow-Suite on the Red Hat 6.2 system (shadow-utils-19990827-10.rpm), but it must be installed manually after installation of the Red Hat software. Versions of Linux available from other vendors may install with the Shadow-Suite. The availability of an rpm for Shadow-Suite may have been the reason why the Shadow-Suite version of login was modified in the lrk4 rootkit as opposed to modifying PAM and the login code that works with it.

## **A.2 The Linux Rootkit V (lrk5)**

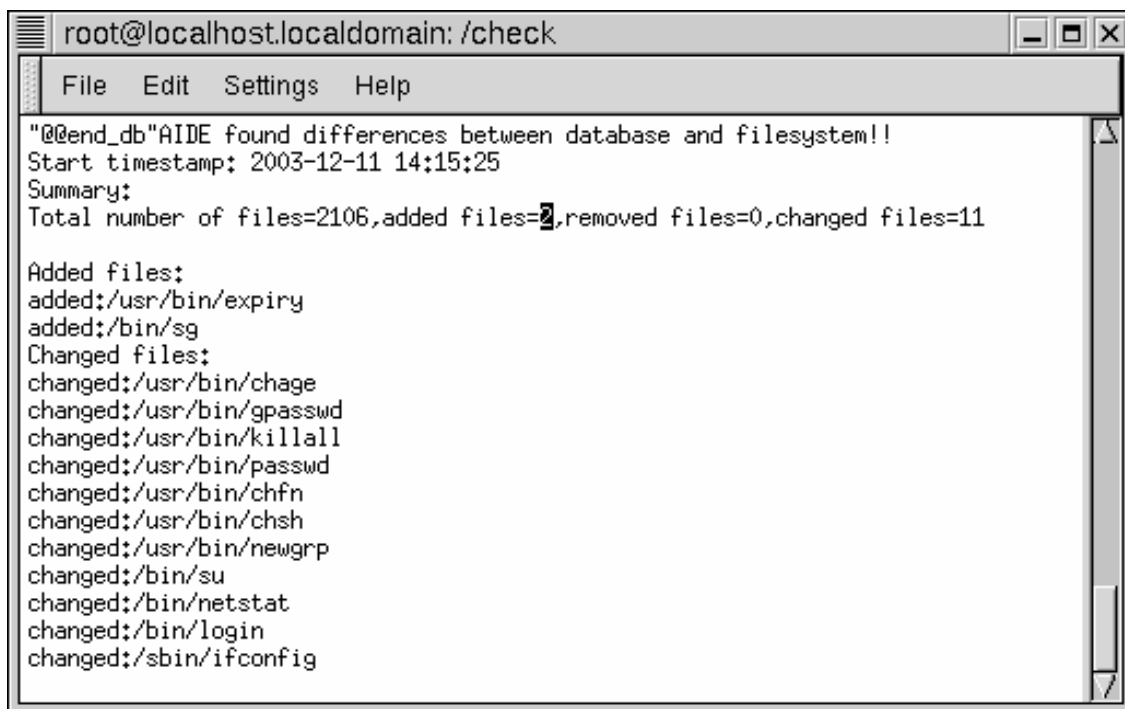
The Linux Rootkit V (lrk5) was released by the same author as lrk4, Lord Somer. In the README file for this rootkit, Lord Somer makes reference to hacked versions of the *ssh* daemon and the *su* utility [65]. This rootkit has been described as one of the most fully featured rootkits presently available [66]. Analysis of the source code indicated that it is based on the previously developed lrk4 rootkit. Like lrk4, this rootkit also utilizes the Shadow Suite of software.

We once again followed our methodology in conducting an analysis of this rootkit. The following steps were taken prior to infecting a system with this rootkit:

1. The file integrity checker program AIDE was run on this system
2. A copy of the kernel text code was produced.
3. The known rootkit detection program, chkrootkit, was run on this system.

All of these steps were taken prior to infecting this system with the lrk5 rootkit. At this point the lrk5 rootkit was installed on the target system. The shadow password installation was utilized during this installation.

The AIDE program detected that 11 files had been changed on this system after the installation of the lrk5 rootkit. Only 8 files were detected as being changed upon installation of the lrk4 rootkit. The chkrootkit program only detected that five files had been infected. The six files that were not detected by the chkrootkit program are the *netstat* file, *chsh* file (both results similar to lrk4) and the *chage*, *gpasswd*, *newgrp*, and *su*. The AIDE program also detected that two files are added to the system by the lrk5 rootkit. These files are: */usr/bin/expiry* and */bin/sg*. Both of these new files are potential signatures that can be used to detect installation of the lrk5 rootkit. Figure 49 shows the changed files that are detected by the AIDE program.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the AIDE program's results: 'AIDE found differences between database and filesystem!!', 'Start timestamp: 2003-12-11 14:15:25', 'Summary:', 'Total number of files=2106,added files=2,removed files=0,changed files=11'. It then lists 'Added files:' as '/usr/bin/expiry' and '/bin/sg', and 'Changed files:' as '/usr/bin/chage', '/usr/bin/gpasswd', '/usr/bin/killall', '/usr/bin/passwd', '/usr/bin/chfn', '/usr/bin/chsh', '/usr/bin/newgrp', '/bin/su', '/bin/netstat', '/bin/login', and '/sbin/ifconfig'.

```
root@localhost.localdomain: /check
File Edit Settings Help
"@@end_db"AIDE found differences between database and filesystem!!
Start timestamp: 2003-12-11 14:15:25
Summary:
Total number of files=2106,added files=2,removed files=0,changed files=11

Added files:
added:/usr/bin/expiry
added:/bin/sg
Changed files:
changed:/usr/bin/chage
changed:/usr/bin/gpasswd
changed:/usr/bin/killall
changed:/usr/bin/passwd
changed:/usr/bin/chfn
changed:/usr/bin/chsh
changed:/usr/bin/newgrp
changed:/bin/su
changed:/bin/netstat
changed:/bin/login
changed:/sbin/ifconfig
```

**Figure 49: AIDE results on lrk5 infected system**

Like the lrk4 rootkit, the lrk5 rootkit did not compromise the integrity of the kernel. The kern\_check program did not detect the redirection of any system calls. A

comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched.

We then examine the six files that were not detected by the chkrootkit program to identify signatures that can be utilized for detection. Using the previously developed methods we identify the following potential signatures:

/usr/bin/chsh	“login.defs”	(same signature as lrk4)
/usr/bin/chage	“etc/spwd”	
/usr/bin/gpasswd	“etc/grp”	
/usr/bin/newgrp	“Sorry”	
/bin/su	“etc/porttime”	
/bin/netstat	“written”	(same signature as lrk4)

We compared the string output of several of the lrk5 modified files to the same modified file from lrk4 for comparison. These files matched. Based on this analysis we determine that the lrk5 rootkit is a modification to the lrk4 Trojan Utility rootkit as described in section 2.3.

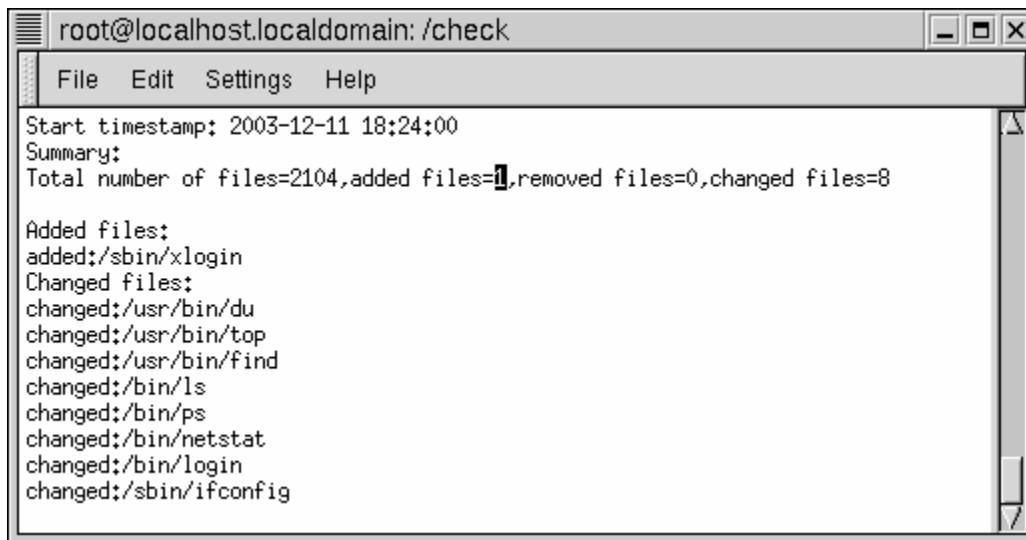
### **A.3 The t0rn rootkit**

The t0rn rootkit was designed to be quickly installed on a vulnerable target system without requiring a high level of skill. This rootkit, unlike lrk4 and lrk5 has the ability to match the timestamp of the original files that it replaces with trojan binary files [67]. It was only available in a precompiled version so there is no source code currently available for analysis. The author of the t0rn rootkit was arrested in the United Kingdom in a Joint Scotland Yard/FBI operation [68].

The code for the t0rn rootkit was incorporated into the li0n worm [69]. The quick and easy installation of the t0rn rootkit most likely led to its incorporation into an automated worm type attack. This worm was very successful in infecting vulnerable systems on the Internet [70].

We continued to follow our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program. Unlike the lrk4 and lrk5 rootkits there was no option to use the shadow passwords.

The AIDE program detected that eight files have been changed on this system after the installation of the t0rn rootkit. Eight files were also detected as being changed upon installation of the lrk4 rootkit. However, only four files match as being changed between these two rootkits. These files are *find*, *netstat*, *login*, and *ifconfig*. The files that do not match are *du*, *top*, *ls*, and *ps*. The AIDE program also detected that the file */sbin/xlogin* has been added to the system. The chkrootkit program only detected that only three files had been infected. These files are *ifconfig*, *login*, and *ps*. The five files that were not detected by the chkrootkit program are *netstat*, *du*, *top*, *ls*, *ps*, and *find*. The chkrootkit program does detect the presence of the t0rn rootkit on the target system. It accomplishes this by searching for the file */sbin/xlogin*. The chkrootkit program also checks for any directories named “.puta” which are directories that are created upon installation of the t0rn rootkit. This is the file that is detected as being added to the system by the AIDE program. Figure 50 shows the changed files that are detected by the AIDE program.



```
root@localhost.localdomain: /check
File Edit Settings Help
Start timestamp: 2003-12-11 18:24:00
Summary:
Total number of files=2104,added files=1,removed files=0,changed files=8

Added files:
added:/sbin/xlogin
Changed files:
changed:/usr/bin/du
changed:/usr/bin/top
changed:/usr/bin/find
changed:/bin/ls
changed:/bin/ps
changed:/bin/netstat
changed:/bin/login
changed:/sbin/ifconfig
```

**Figure 50: AIDE results on t0rn infected system**

Like the lrk4 and lrk5 rootkit, the t0rn rootkit did not compromise the integrity of the kernel. The kern\_check program did not detect the redirection of any system calls. A comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched.

We then examine the five files that were not detected by the chkrootkit program to identify signatures that can be utilized for detection. Using the previously developed methods we identified the following potential signatures:

/usr/bin/du	“.puta”
/usr/bin/top	“.puta”
/usr/bin/find	“.puta”
bin/ls	“.puta”
/bin/netstat	“.puta”

It is significant to note that the same string signature can be used to detect all five of these files that are compromised by the t0rn rootkit. Analysis of the t0rn rootkit indicated

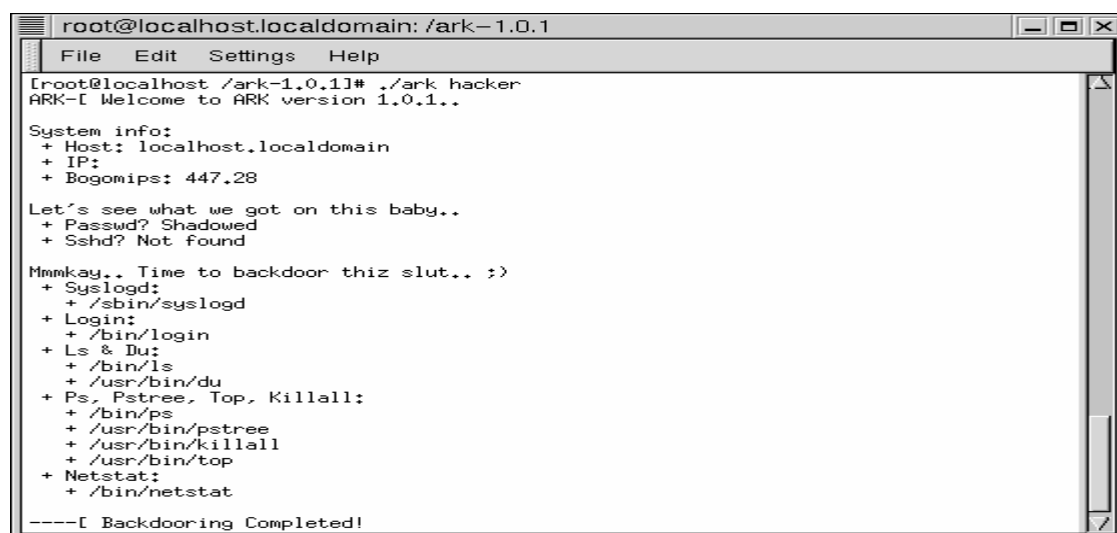
that .puta is a hidden directory that is created and used by this rootkit.

We compared the string output of several of the t0rn modified files to the same modified file from lrk4 for comparison. These files did not match. Based on this analysis we determine that the torn rootkit is not related to lrk4 or lrk5.

## A.4 The ark rootkit

The Ambient Rootkit for Linux, or ark, is available for download from the Internet. This rootkit, like the t0rn rootkit was designed to be quickly installed on a vulnerable target system without requiring a high level of skill. This rootkit, unlike the t0rn rootkit, does not have the ability to match the timestamp of the original files that it replaces with trojan binary files. It was only available in a precompiled version so there is no source code currently available for analysis. There is no documentation concerning this rootkit currently available on the Internet.

Figure 52 shows the install script for this rootkit on the target system. The README file for this rootkit does not include installation instructions.



```
root@localhost.localdomain: /ark-1.0.1
File Edit Settings Help
[root@localhost /ark-1.0.1]# ./ark hacker
ARK-[ Welcome to ARK version 1.0.1..

System info:
+ Host: localhost.localdomain
+ IP:
+ Bogomips: 447.28

Let's see what we got on this baby..
+ Passwd? Shadowed
+ Sshd? Not found

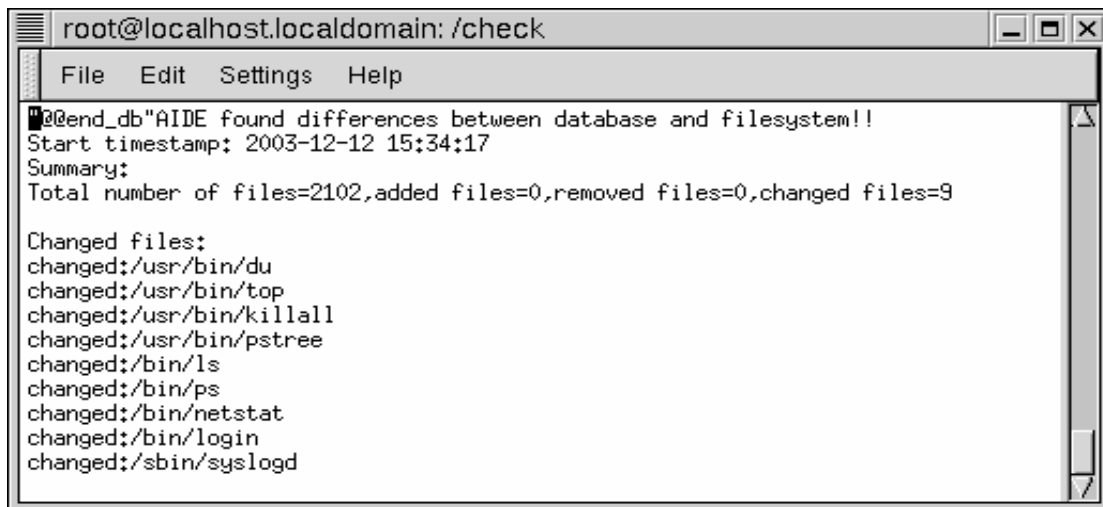
Mmmkay.. Time to backdoor thiz slut.. ;)
+ Syslogd:
+ /sbin/syslogd
+ Login:
+ /bin/login
+ Ls & Du:
+ /bin/ls
+ /usr/bin/du
+ Ps, Pstree, Top, Killall:
+ /bin/ps
+ /usr/bin/pstree
+ /usr/bin/killall
+ /usr/bin/top
+ Netstat:
+ /bin/netstat

----[ Backdooring Completed!
```

Figure 51: ark rootkit installation script

We continued to follow our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program. The ark rootkit contains executable files to address both the use of normal and shadow passwords (*login-normal* and *login-shadow*). The install script was able to make a determination to use the *login-shadow* file for the target system. This is indicated by the line “Passwd? Shadowed” in Figure 51.

The AIDE program detected that nine files had been changed on this system after the installation of the ark rootkit. These same nine files are indicated as being replaced by the ark install script. The chkrootkit program detected all nine of the files that were replaced by the ark rootkit on the target system. In addition, the chkrootkit program does detect the presence of the ark rootkit on the target system. It accomplishes this by searching for the directory */dev/ptyxx*. Figure 52 shows the changed files that are detected by the AIDE program.

A screenshot of a terminal window titled 'root@localhost.localdomain: /check'. The window has a menu bar with 'File', 'Edit', 'Settings', and 'Help'. The terminal output shows the AIDE program's results: '@@end\_db"AIDE found differences between database and filesystem!!', 'Start timestamp: 2003-12-12 15:34:17', 'Summary:', 'Total number of files=2102,added files=0,removed files=0,changed files=9', and a list of 'Changed files:' including '/usr/bin/du', '/usr/bin/top', '/usr/bin/killall', '/usr/bin/pstree', '/bin/ls', '/bin/ps', '/bin/netstat', '/bin/login', and '/sbin/syslogd'.

```
root@localhost.localdomain: /check
File Edit Settings Help
@@end_db"AIDE found differences between database and filesystem!!
Start timestamp: 2003-12-12 15:34:17
Summary:
Total number of files=2102,added files=0,removed files=0,changed files=9

Changed files:
changed:/usr/bin/du
changed:/usr/bin/top
changed:/usr/bin/killall
changed:/usr/bin/pstree
changed:/bin/ls
changed:/bin/ps
changed:/bin/netstat
changed:/bin/login
changed:/sbin/syslogd
```

**Figure 52: AIDE results from an ark infected system**



Like the t0rn rootkit, the ark rootkit did not compromise the integrity of the kernel. The kern\_check program did not detect the redirection of any system calls. A comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched.

This is the first case that we have analyzed in which the chkrootkit program was able to detect all of the files that were listed as being changed by the AIDE File Integrity Checker program. It is not necessary to produce any new string signatures for the ark rootkit. A system administrator could follow this methodology to ensure that the chkrootkit program was capable of detecting all files modified on any suspect systems by a rootkit such as ark, that is, the system administrator would not have to produce any additional signatures that needed to be checked.

Comparison of the executable replacement files with the executable replacement files from the t0rn rootkit as well as lrk4 and lrk5 do not indicate that any of these files are similar. However, analysis of the *login* program from the ark rootkit indicates that it is based on a later version of the Shadow-Suite of software than was used to develop the lrk4 and lrk5 rootkits. Therefore, ark may be a modification to the lrk4 and lrk5 rootkits.

## **A.5 The knark rootkit**

An initial analysis of knark was conducted by Toby Miller in 2001 [71]. The knark rootkit is a kernel level rootkit as described in section 2.4.1. Various versions for knark are available on the Internet. We chose to examine version 0.59, which is the last known version to have been authored by CREED. The version of the rootkit can be installed on the Linux 2.2 kernel but will not install on the Linux 2.4 kernel. CREED is identified as the original author of knark. This version of knark can be downloaded from the

packetstorm web site [72]. There is also an interview with CREED that is available for download on the web [73].

Another analysis of knark was conducted by Jonathan Clemmens [75]. In this analysis, the link is made between knark and an earlier kernel level rootkit known as “heroin.c”. The link to “heroin.c” is also made in the interview with CREED. The following are recognized “features” of the knark kernel level rootkit:

- Hide/Unhide files or directories
- Hide TCP/UDP connections
- Execution Redirection
- Unauthenticated privilege escalation via the *rootme* program within knark
- Ability to change UID/GID of a running process
- Unauthenticated, privileged remote execution daemon
- Kill -31 to hide a running process

The uses of file hiding and execution redirection allow a hacker to install a backdoor on the target system that may not be detected through conventional system analysis.

We once again followed our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program.

The AIDE program did not detect any file modifications after the knark rootkit was installed on the target system. The chkrootkit program did detect the presence of the knark rootkit. This program looks for the creation of certain directories as a result of a system being infected with a rootkit. The knark kernel rootkit creates a directory called

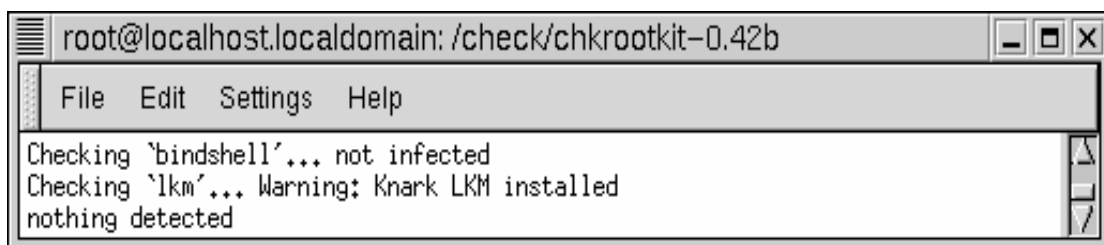
knark in the /proc/ directory. A system infected with the knark kernel level rootkit can be detected by chkrootkit because of the presence of this directory. We speculate that the developer of knark, CREED, specifically chose the /proc/ directory for the sub-directory that is to be created. The /proc/ directory is one that is constantly changing during the operation of the computer. Any new process that is started will have a separate directory created here. Because of this, the /proc/ directory is normally not chosen as a directory that is checked with a cryptographic signature. This hidden directory is created by knark and the chkrootkit program looks for this specific directory. chkrootkit also checks the system logs. Therefore the signature must be known for chkrootkit to detect if a rootkit has been installed on a system. The code in the chkrootkit script file that is used to detect an infection by knark is shown in Figure 53.

```
## knark LKM

if [ -d /proc/knark ]; then
    echo "Warning: Knark LKM installed"
fi
```

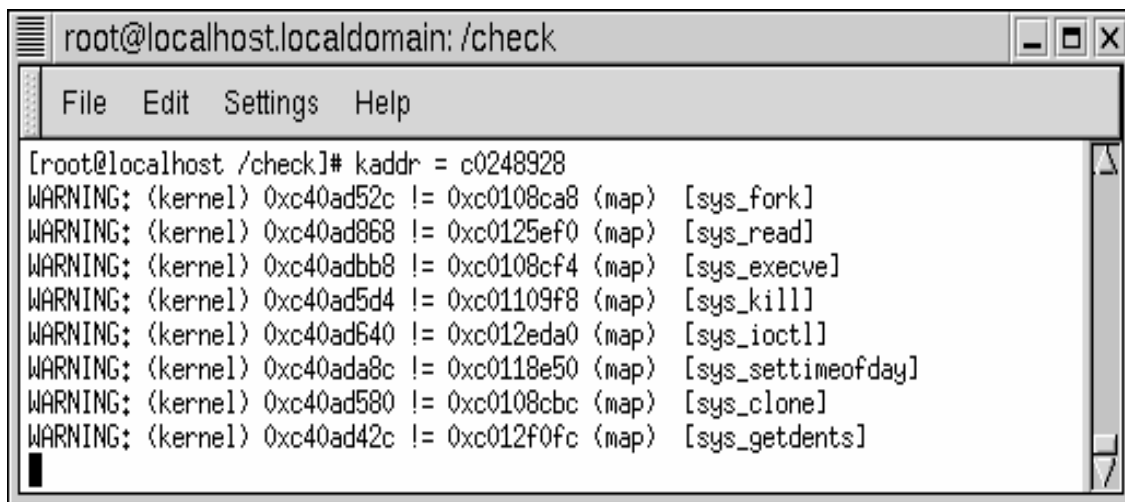
**Figure 53: chkrootkit code to detect knark lkm**

We find it unusual that the knark rootkit does not use its directory hiding capability to hide this directory. The chkrootkit check can be defeated if this directory is renamed to something besides /proc/knark. Figure 54 shows the output of the chkrootkit program on a system that is infected with the knark rootkit.



**Figure 54: chkrootkit output of knark infected system**

The kern\_check program does detect that eight system calls have been modified on the target system. The output indicates that the addresses of these eight sys\_calls currently listed in the sys\_call\_table do not match the addresses for those sys\_calls in the original map of the kernel symbols. This map of kernel systems is available on the system as /boot/System.map. It is theoretically possible for another kernel level rootkit to change the same eight system calls. Figure 55 shows the results of running the kern\_check program on the target system.



```
root@localhost.localdomain: /check
File Edit Settings Help
[root@localhost /check]# kaddr = c0248928
WARNING: (kernel) 0xc40ad52c != 0xc0108ca8 (map) [sys_fork]
WARNING: (kernel) 0xc40ad868 != 0xc0125ef0 (map) [sys_read]
WARNING: (kernel) 0xc40adbb8 != 0xc0108cf4 (map) [sys_execve]
WARNING: (kernel) 0xc40ad5d4 != 0xc01109f8 (map) [sys_kill]
WARNING: (kernel) 0xc40ad640 != 0xc012eda0 (map) [sys_ioctl]
WARNING: (kernel) 0xc40ada8c != 0xc0118e50 (map) [sys_settimeofday]
WARNING: (kernel) 0xc40ad580 != 0xc0108cbc (map) [sys_clone]
WARNING: (kernel) 0xc40ad42c != 0xc012f0fc (map) [sys_getdents]
```

**Figure 55: kern\_check results on knark infected system**

These results show the danger of kernel rootkits such as knark. File integrity checker programs such as AIDE will not detect any modifications on the target system. Programs such as chkrootkit can be defeated if a known signature is not available.

Our methodology depends on the ability to archive a copy of the system call code that currently exists in kernel memory for characterization analysis. It is this archived code that we use to be able to characterize kernel rootkit exploits. We have developed a C program that can copy the system call code that is referenced by a start and end address

and write the executable object code to a file for future reference. We feel that this is significant because it allows the analyst to be able to retrieve of the code that is currently running in the system kernel. Further, some types of kernel level rootkits such as knark do not remain resident in memory after the system is rebooted. Our program allows for a copy of any suspicious system calls to be copied offline for follow on analysis prior to rebooting the system.

Analysis of the source code used to create the knark rootkit indicated that the new redirected system calls were being written sequentially into kernel memory. This may not always be the case and it may be necessary to conduct an analysis of the object code to identify start and end address of the individual system calls.

The archived files can be examined with a tool such as binary visual editor (*bvi*) which is available on the Internet [75]. The output of *bvi* is the addresses of the data relative to the beginning of the file (far left), the actual data in hexadecimal notation (center), and the data in ASCII format (far right) as indicated in Figure 56. One can search within the file hexadecimal notation for the start and end of each system call by looking for the individual opcodes for pushing and popping the registers (each system call is a separate C code routine that will push and pop values on to the stack. You can also identify the end of each system call routine by looking for the one byte return opcode (ret – C3 in the Intel x86 architecture [76]).

Figure 56 shows us the *bvi* output for the knark\_getdents system call that replaces the original sys\_getdents system call. This system call is used by the kernel to output the contents of a directory. Kernel level rootkits may choose to compromise this system call to hide files and directories on the target system. The binary *ls* command on the system

that has the rootkit installed does not have to be modified in this case.

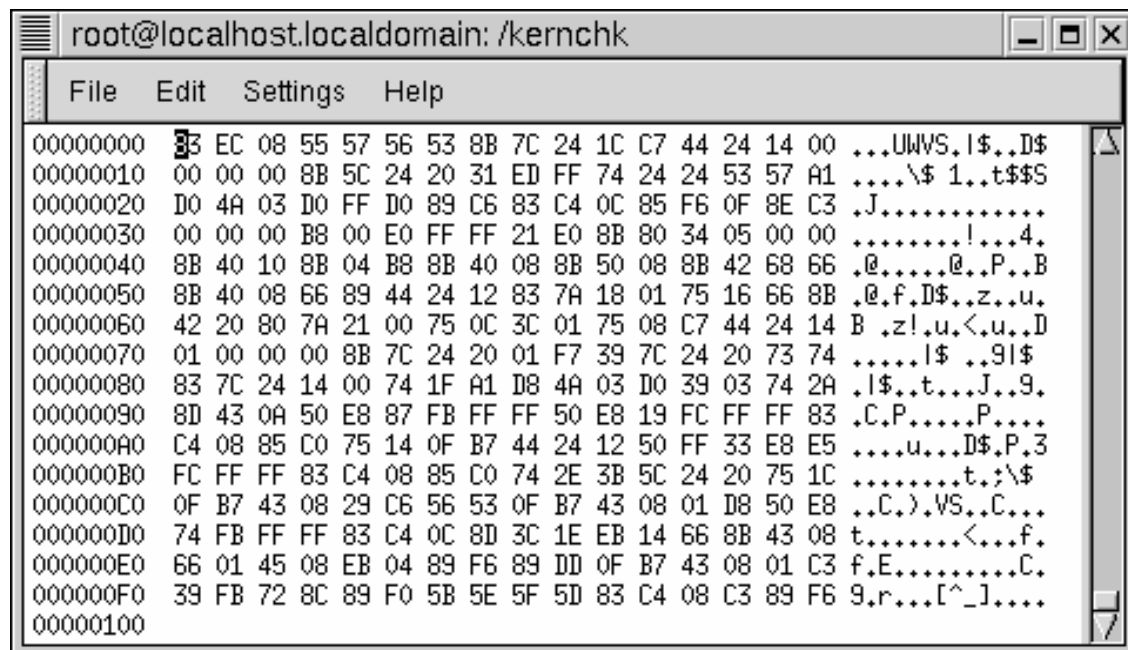


Figure 56: *bvi* analysis of *getdents* system call

The archived system calls that are installed by the rootkit can be used by system administrators and security personnel to develop a fingerprint for a particular kernel rootkit such as knark. A comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched. This indicates that knark did not modify the kernel text segment of code and indicates that knark is a kernel level rootkit that modifies the `sys_call_table`, as described in section 2.4.1. The `sys_call_table` is not maintained within the kernel text segment of kernel memory.

## A.6 The adore rootkit

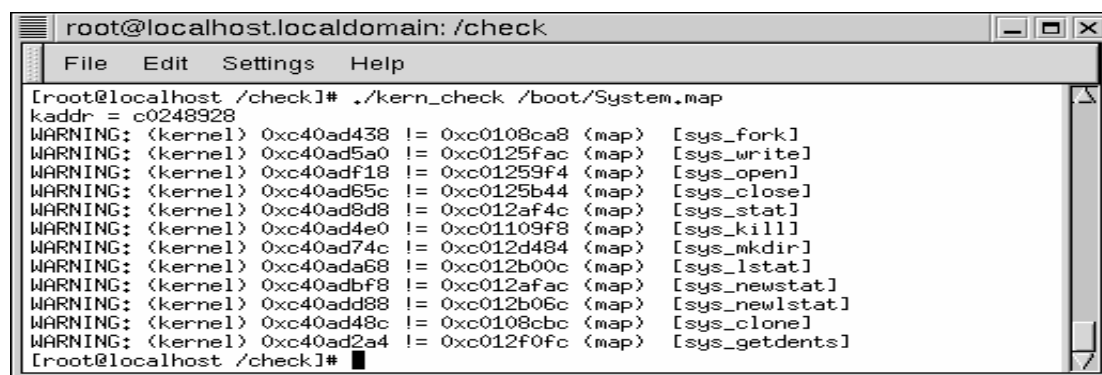
The adore rootkit is described as a Linux LKM rootkit that is easy to install. It only requires minor configuration adjustments. The adore kernel rootkit has the capability to

hide files and PID's. This rootkit will also establish a backdoor port for the hacker to connect to upon installation. It is also possible to uninstall adore using the *ava* utility that installs with the rootkit. [77]

We once again followed our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program.

The AIDE program did not detect any file modifications after the adore rootkit was installed on the target system. The chkrootkit program did not detect the presence of the adore rootkit. At present there are no signatures available to the chkrootkit developers to detect adore. Unlike knark, adore does not create any files or directories on the target system.

The kern\_check program does detect that twelve system calls are modified on the target system. The output indicates that the addresses of these twelve sys\_calls currently listed in the sys\_call\_table do not match the addresses for those sys\_calls in the original map of the kernel symbols (/boot/System.map). Figure 57 shows the results of running the kern\_check program on the target system.



```
root@localhost.localdomain: /check
File Edit Settings Help
[root@localhost /check]# ./kern_check /boot/System.map
kaddr = c0248928
WARNING: (kernel) 0xc40ad438 != 0xc0108ca8 (map) [sys_fork]
WARNING: (kernel) 0xc40ad5a0 != 0xc0125fac (map) [sys_write]
WARNING: (kernel) 0xc40adf18 != 0xc01259f4 (map) [sys_open]
WARNING: (kernel) 0xc40ad65c != 0xc0125b44 (map) [sys_close]
WARNING: (kernel) 0xc40ad8d8 != 0xc012af4c (map) [sys_stat]
WARNING: (kernel) 0xc40ad4e0 != 0xc01109f8 (map) [sys_kill]
WARNING: (kernel) 0xc40ad74c != 0xc012d484 (map) [sys_mkdir]
WARNING: (kernel) 0xc40ada68 != 0xc012b00c (map) [sys_lstat]
WARNING: (kernel) 0xc40adbfb != 0xc012afac (map) [sys_newstat]
WARNING: (kernel) 0xc40add88 != 0xc012b06c (map) [sys_newlstat]
WARNING: (kernel) 0xc40ad48c != 0xc0108cbc (map) [sys_clone]
WARNING: (kernel) 0xc40ad2a4 != 0xc012f0fc (map) [sys_getdents]
[root@localhost /check]#
```

Figure 57: kern\_check results on adore infected system

A comparison was made between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched. This indicates that like knark, adore did not modify the kernel text segment of code and indicates that adore is a kernel level rootkit that modifies the `sys_call_table`, similar to the knark rootkit. This may indicate that knark and adore may be related.

Four of the twelve system calls that are changed by adore are also changed by knark. These four system calls are: `sys_getdents`, `sys_fork`, `sys_clone`, and `sys_kill`. However, a comparison of the kernel code for the adore system calls and the knark system calls indicates that these system calls are different. We retrieved the system call code from within the kernel with the program that we have previously discussed. We therefore conclude that the adore rootkit is different from the knark rootkit.

During our analysis of the adore rootkit we recognized another method to identify the ending address of a system call within kernel memory. If the object file that the kernel rootkit utilizes to install on the target system is still available on that target system, then it may be possible to determine the exact end address of the system call within kernel memory. In the case of the adore rootkit, this file was called the `adore.o` file. The start address of each system call can be determined from the output of the `kern_check` program (see figure 57 concerning adore). We have already discussed use of the *bvi* program to determine the ending address as well as using the output of the `kern_check` program if you can assume that the system calls are written sequentially in kernel space. The exact ending address may be available within the object file. Using a program such as the GNU debugger (*gdb*), it may be possible to disassemble each replaced system call. To do this you must know the name of the replacement system calls within the object file. The



names of the replacement system calls can be determined from any source code files that may be available from the rootkit. Figure 58 shows the results of disassembling an adore system call (sys\_fork).

```

root@localhost.localdomain: /adore
File Edit Settings Help
(gdb) disass n_fork
Dump of assembler code for function n_fork:
0x3e8 <n_fork>: push    %ebp
0x3e9 <n_fork+1>: mov     %esp,%ebp
0x3eb <n_fork+3>: push    %edi
0x3ec <n_fork+4>: push    %esi
0x3ed <n_fork+5>: push    %ebx
0x3ee <n_fork+6>: mov     $0xfffffe000,%eax
0x3f3 <n_fork+11>: and     %esp,%eax
0x3f5 <n_fork+13>: pushl   0x5c(%eax)
0x3f8 <n_fork+16>: call    0x3f9 <n_fork+17>
0x3fd <n_fork+21>: add     $0x4,%esp
0x400 <n_fork+24>: test    %eax,%eax
0x402 <n_fork+26>: setne   %al
0x405 <n_fork+29>: movzbl  %al,%ebx
0x408 <n_fork+32>: add     $0xffffffc4,%esp
0x40b <n_fork+35>: mov     %esp,%edi
0x40d <n_fork+37>: lea     0x8(%ebp),%esi
0x410 <n_fork+40>: cld
0x411 <n_fork+41>: mov     $0xf,%ecx
0x416 <n_fork+46>: repz    movsl %ds:(%esi),%es:(%edi)
0x418 <n_fork+48>: mov     0x0,%eax
0x41d <n_fork+53>: call    *%eax
0x41f <n_fork+55>: mov     %eax,%esi
---Type <return> to continue, or q <return> to quit---
0x421 <n_fork+57>: add     $0x3c,%esp
0x424 <n_fork+60>: test    %ebx,%ebx
0x426 <n_fork+62>: je      0x432 <n_fork+74>
0x428 <n_fork+64>: test    %esi,%esi
0x42a <n_fork+66>: jl      0x432 <n_fork+74>
0x42c <n_fork+68>: push    %esi
0x42d <n_fork+69>: call    0x42e <n_fork+70>
0x432 <n_fork+74>: mov     %esi,%eax
0x434 <n_fork+76>: lea     0xffffffff4(%ebp),%esp
0x437 <n_fork+79>: pop     %ebx
0x438 <n_fork+80>: pop     %esi
0x439 <n_fork+81>: pop     %edi
0x43a <n_fork+82>: leave
0x43b <n_fork+83>: ret
End of assembler dump.
(gdb)

```

Figure 58: gdb output of adore system call

The final statement of this disassembled code is a return statement (ret) which we have previously discussed. According to *gdb*, the `sys_fork` system call replacement, which is named `n_fork` in the *adore* source code is 83 bytes in size. The start address of this system call as indicated by the `kern_check` output is 0xc40ad438. Adding 83 bytes to this address gives an ending address of 0xc40ad48b. The next system call in sequence is the `sys_clone` system call, which starts at address 0xc40ad48c.

## **A.7 The knark rootkit targeting the Linux 2.4 kernel**

The knark rootkit developed by CREED described in Appendix A.5 was targeted against the Linux 2.2 kernel and was not capable of installing on the Linux 2.4 kernel. This rootkit was ported over to the Linux 2.4 kernel by an individual known as Cyberwind. The name of this version of knark is knark-2.4.3 and it is based on knark-0.59 described in A.5.

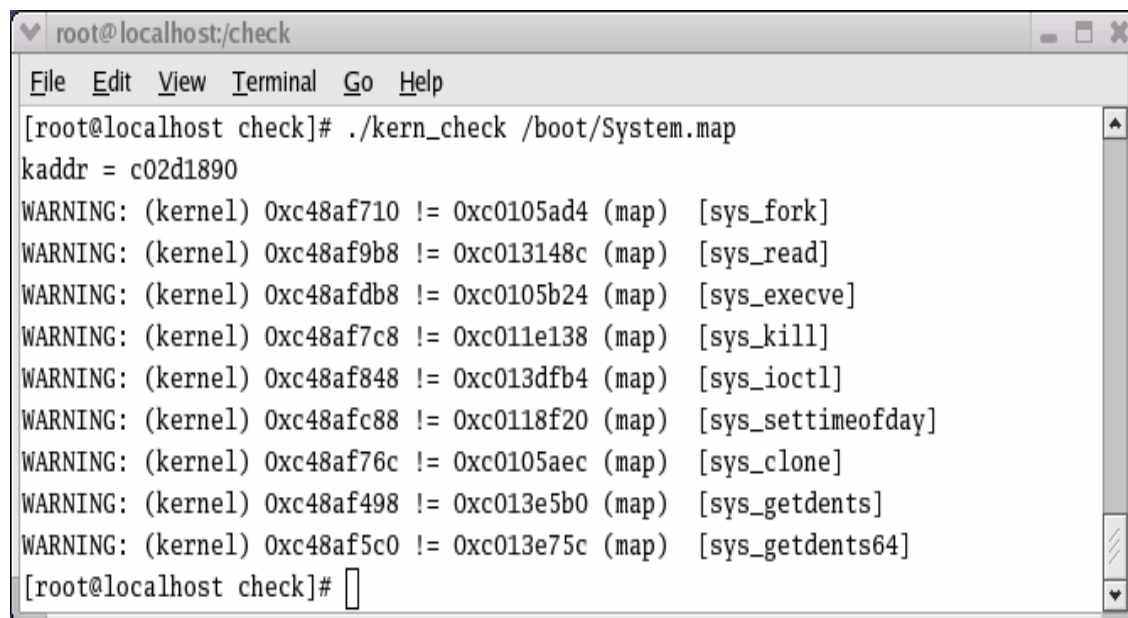
There is a `README.cyberwind` file that describes some of the necessary changes that were required to install this system on the Linux 2.4 kernel. The following changes were required:

1. Rewrite of the `/proc` code since node registration and cleanup differs in the Linux 2.4 kernel from the Linux 2.2 kernel.
2. Linux 2.4 use of `getdents64` to identify a dentry and as a result, `sys_getdents64` must now be intercepted for this kernel rootkit to work as intended.

We once again follow our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the `chkrootkit` program. Like the previous version of knark, the AIDE

program did not detect any file modifications after this new version of the knark rootkit was installed on the target system. The chkrootkit program did detect the presence of this knark rootkit. This version of the knark kernel rootkit also creates a directory called knark in the /proc/ directory. The same check can be made by chkrootkit because of the presence of this same directory. Like the previous version of knark a comparison between the current kernel text segment and the archived kernel text segment that was produced prior to the infection matched.

This version of knark changes nine system calls. The earlier version of knark changed eight system calls. The newer version on knark changes the same eight system calls as the previous version of knark and also changes the sys\_getdents64 system call. The reason for changing this additional system call is explained in the README.cybersinds file. Figure 59 shows the output of the kern\_check program on a system that is infected with the knark rootkit.



```
root@localhost:/check
File Edit View Terminal Go Help
[root@localhost check]# ./kern_check /boot/System.map
kaddr = c02d1890
WARNING: (kernel) 0xc48af710 != 0xc0105ad4 (map) [sys_fork]
WARNING: (kernel) 0xc48af9b8 != 0xc013148c (map) [sys_read]
WARNING: (kernel) 0xc48afdb8 != 0xc0105b24 (map) [sys_execve]
WARNING: (kernel) 0xc48af7c8 != 0xc011e138 (map) [sys_kill]
WARNING: (kernel) 0xc48af848 != 0xc013dfb4 (map) [sys_ioctl]
WARNING: (kernel) 0xc48afc88 != 0xc0118f20 (map) [sys_settimeofday]
WARNING: (kernel) 0xc48af76c != 0xc0105aec (map) [sys_clone]
WARNING: (kernel) 0xc48af498 != 0xc013e5b0 (map) [sys_getdents]
WARNING: (kernel) 0xc48af5c0 != 0xc013e75c (map) [sys_getdents64]
[root@localhost check]#
```

Figure 59: kern\_check results on system infected with knark for Linux 2.4

Analysis of these changed system calls using the kdb program listed the actual replacement system call labels. Using the start addresses produced by the kern\_check program made it possible to find the exact end point of each replacement system call even if the object file were not available. This may be a result of how the knark program is compiled on this kernel or the newer version of kdb that is run on the Linux 2.4 kernel. Figure 60 shows the results of running kdb in this fashion.

```
kdb>id 0xc48af5bb -approximate endpoint of knark_getdents
0xc48af5bb knark_getdents+0123: jmp 0xc48af4fc
0xc48af5c0 knark_getdents64:      push %ebp
```

**Figure 60: kdb output of system infected with knark 2.4.3**

The second line of output from the kdb program is the first line of the knark\_getdents64 replacement system call. This address matches the address of the replacement system call as indicated by the kern\_check program (Figure 59).

This version of knark (knark-2.4.3) is based on a previous version (knark-0.59) and is a modification to that version. However, the kernel code copies of the eight modified system calls changed by both versions do not match. This is most likely due to the fact that each version is installed on a different kernel. This demonstrates that individuals following this methodology must be sure to use similar systems in their analysis.

## **A.8 The SuckIT toolkit**

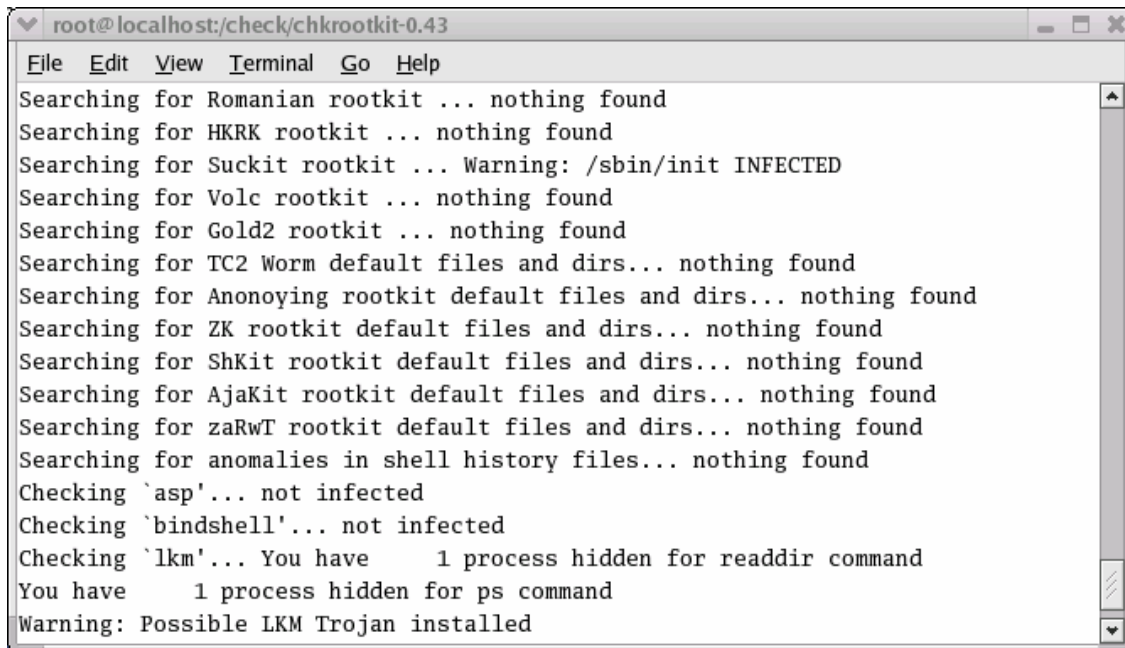
The SuckIT toolkit was developed by sd and devik based on the article they wrote in PHRACK vol. 58, article 7, titled “Linux-on-the-fly kernel patching without LKM”. This article discusses a methodology for modifying the system calls within the Linux

kernel without the use of LKM support or the /boot/System.map file [78]. Unlike kernel level rootkits that modify the system call table, this type of rootkit keeps the original system call table intact. An examination of the original system call table will not indicate that the system has been compromised by a kernel level rootkit. The SuckIT kernel level rootkit accomplishes this by modifying the System Call Interrupt (system\_call() function) that is triggered whenever a User Mode process invokes a system call [79]. The pointer to the normal system call table is changed to the address of the new system call table that is created by the SuckIT rootkit. This new system call table contains the addresses of the malicious system calls that are modified by the SuckIT rootkit as well as the original addresses of any unmodified system calls. Our methodology retrieves the address of the system call table that is stored within the System Call Interrupt and checks this table for modifications. Any modification to this table as well as a mismatch between this retrieved address and the address of the system call table that is maintained within the /boot/System.map file will also indicate that redirection of the system call table is occurring within the kernel.

We continued to follow our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program.

The AIDE program did not detect any file modifications after the SuckIT rootkit was installed on the target system. However, the AIDE program does detect that one file now has attributes that differ from the attributes that are stored in the original AIDE database. The file in question is the /sbin/telinit file. The most recent version of the chkrootkit

program (released on 27 DEC 2003) did detect the presence of the SuckIT rootkit. This new version of chkrootkit detects that the /sbin/init file is infected by checking for the presence of the text string “init.” within the /proc/1/maps file. This file contains lists of libraries, executables, and other files that make up the text image of the init process that is currently running on the target system [80]. Figure 61 shows the output of the chkrootkit program on a system that is infected with the SuckIT rootkit. The third line of output shows that the /sbin/init file has been modified. It is significant to note that other rootkits that are based on SuckIT will most likely trigger the same indication.

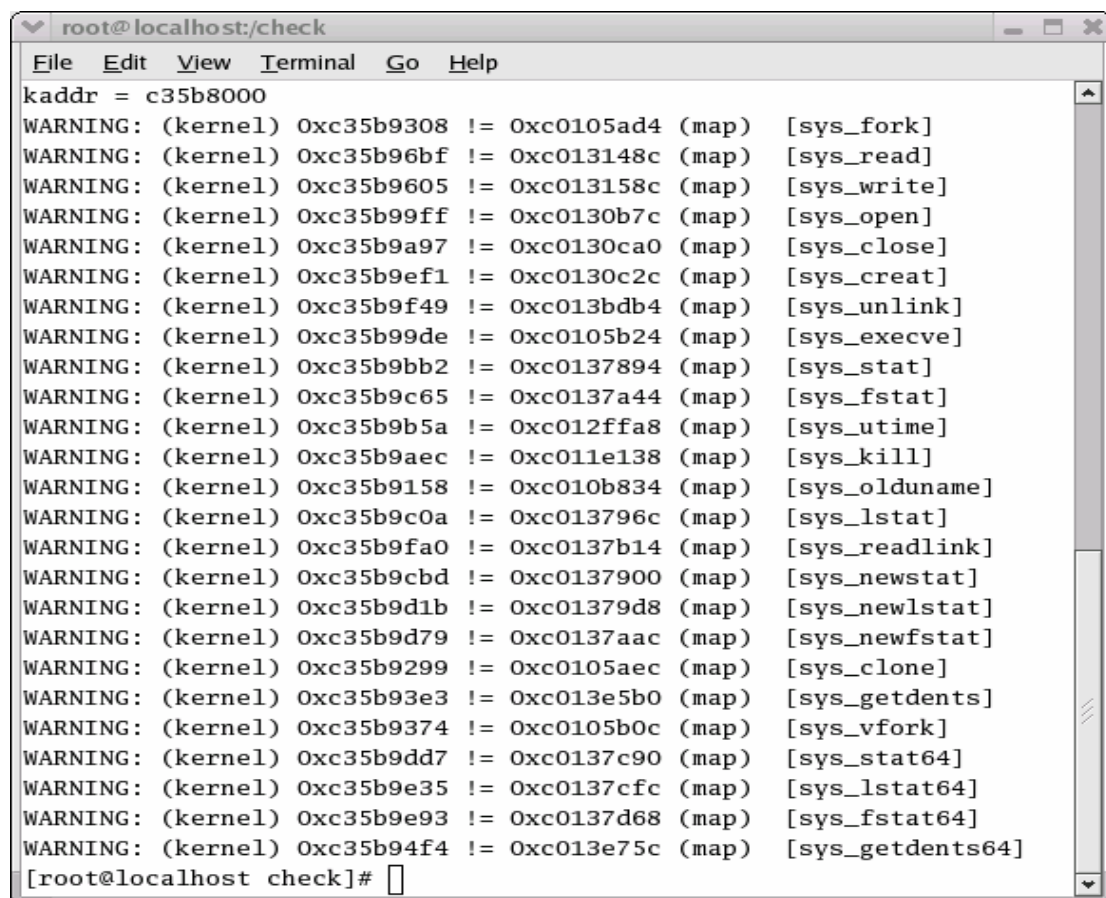


```
root@localhost:/check/chkrootkit-0.43
File Edit View Terminal Go Help
Searching for Romanian rootkit ... nothing found
Searching for HKRK rootkit ... nothing found
Searching for Suckit rootkit ... Warning: /sbin/init INFECTED
Searching for Volc rootkit ... nothing found
Searching for Gold2 rootkit ... nothing found
Searching for TC2 Worm default files and dirs... nothing found
Searching for Anonoying rootkit default files and dirs... nothing found
Searching for ZK rootkit default files and dirs... nothing found
Searching for ShKit rootkit default files and dirs... nothing found
Searching for AjaKit rootkit default files and dirs... nothing found
Searching for zaRwT rootkit default files and dirs... nothing found
Searching for anomalies in shell history files... nothing found
Checking `asp'... not infected
Checking `bindshell'... not infected
Checking `lkm'... You have      1 process hidden for readdir command
You have      1 process hidden for ps command
Warning: Possible LKM Trojan installed
```

**Figure 61: chkrootkit detecting SuckIT**

The SuckIT rootkit changes 25 system calls. This is more than any previous kernel rootkit that we have examined. The ktext files now differ on a target system that is infected with SuckIT. In fact, once a system is infected with SuckIT, every subsequent ktext file will differ from its previous version upon system reboot. This is because

SuckIT is resident in memory and will modify each kernel text segment in a different fashion. Unlike knark and adore, the SuckIt rootkit does not modify the original system call table of addresses that is maintained within kernel memory. Instead, SuckIT creates an entirely new instance of the system call table in kernel memory and redirects all subsequent system call references to this new table. Upon reboot this new system call table is placed at a different address in kernel memory causing the ktext file to differ. This type of rootkit was described in section 2.4.2. It is necessary to retrieve the address of this newly created system call table to check the validity of the system calls. Figure 62 shows the results of checking the system call table that is used by a target system infected with the SuckIT rootkit.



```
root@localhost:/check
File Edit View Terminal Go Help
kaddr = c35b8000
WARNING: (kernel) 0xc35b9308 != 0xc0105ad4 (map) [sys_fork]
WARNING: (kernel) 0xc35b96bf != 0xc013148c (map) [sys_read]
WARNING: (kernel) 0xc35b9605 != 0xc013158c (map) [sys_write]
WARNING: (kernel) 0xc35b99ff != 0xc0130b7c (map) [sys_open]
WARNING: (kernel) 0xc35b9a97 != 0xc0130ca0 (map) [sys_close]
WARNING: (kernel) 0xc35b9ef1 != 0xc0130c2c (map) [sys_creat]
WARNING: (kernel) 0xc35b9f49 != 0xc013bdb4 (map) [sys_unlink]
WARNING: (kernel) 0xc35b99de != 0xc0105b24 (map) [sys_execve]
WARNING: (kernel) 0xc35b9bb2 != 0xc0137894 (map) [sys_stat]
WARNING: (kernel) 0xc35b9c65 != 0xc0137a44 (map) [sys_fstat]
WARNING: (kernel) 0xc35b9b5a != 0xc012ffa8 (map) [sys_utime]
WARNING: (kernel) 0xc35b9aec != 0xc011e138 (map) [sys_kill]
WARNING: (kernel) 0xc35b9158 != 0xc010b834 (map) [sys_olduname]
WARNING: (kernel) 0xc35b9c0a != 0xc013796c (map) [sys_lstat]
WARNING: (kernel) 0xc35b9fa0 != 0xc0137b14 (map) [sys_readlink]
WARNING: (kernel) 0xc35b9cbd != 0xc0137900 (map) [sys_newstat]
WARNING: (kernel) 0xc35b9d1b != 0xc01379d8 (map) [sys_newlstat]
WARNING: (kernel) 0xc35b9d79 != 0xc0137aac (map) [sys_newfstat]
WARNING: (kernel) 0xc35b9299 != 0xc0105aec (map) [sys_clone]
WARNING: (kernel) 0xc35b93e3 != 0xc013e5b0 (map) [sys_getdents]
WARNING: (kernel) 0xc35b9374 != 0xc0105b0c (map) [sys_vfork]
WARNING: (kernel) 0xc35b9dd7 != 0xc0137c90 (map) [sys_stat64]
WARNING: (kernel) 0xc35b9e35 != 0xc0137cfc (map) [sys_lstat64]
WARNING: (kernel) 0xc35b9e93 != 0xc0137d68 (map) [sys_fstat64]
WARNING: (kernel) 0xc35b94f4 != 0xc013e75c (map) [sys_getdents64]
[root@localhost check]#
```

Figure 62: kern\_check results on system infected with SuckIT

Our examination of the SuckIT rootkit revealed to us the first difference, or  $\nabla$  in functionality between SuckIT and the program that it replaces. SuckIT overwrites a location in kernel memory that contains the address of the system call table. SuckIT is able to accomplish this by querying a specific register within the processor. It then use this information to find the entry point address within the kernel for the system call table and overwrites this address with the address of a new system call table containing the addresses of some malicious system calls that SuckIT also creates.

One of the key features of the SuckIT rootkit is its ability to identify the correct location to overwrite within the kernel memory. The SuckIT rootkit uses the following segment of code within the `install.c` program file to do this:

```
asm ("sidt %0" : "=m" (idtr));

printf("RK_Init: idt=0x%08x, ", (uint) idtr.base);

if (ERR(rkm(fd, &idt80, sizeof(idt80),
        idtr.base + 0x80 * sizeof(idt80)))) {
    printf("IDT table read failed (offset
        0x%08x)\n",
        (uint) idtr.base);
    close(fd);
    return 1;
}

old80 = idt80.off1 | (idt80.off2 << 16);
sct = get_sct(fd, old80, sctp);
```

This code works by querying the processor for the address of the Interrupt Descriptor Table. The SuckIT program uses the `sidt` command to accomplish this. The `sidt` command is part of the Instruction Set for the INTEL Pentium (x86) Architecture. The purpose of this command is to store the Interrupt Descriptor Table Register (`idtr`) in the



destination operand [81]. A different command would be required if Linux were implemented on an architecture that differed from the INTEL Pentium (x86) architecture. SuckIT was written to run on this architecture. This rootkit first makes use of the `asm("sidt %0 : "=m" (idtr));` command. The `asm` command signifies to the compiler that assembly language instructions are being used. This command returns the address of the Interrupt Descriptor Table within kernel memory. This address is then printed out by the `printf("RK_Init: idt=0x%08x, ", (uint) idtr.base);` command. The next series of commands is where the program retrieves the actual address of the System Call Interrupt (`system_call()` function) from the Interrupt Descriptor Table. To invoke this function within Linux, the `int $0x80` assembly instruction must be invoked. The `install.c` program calls a function `rkm` that reads kernel memory with the following line of code:

```
rkm(fd,&idt80,sizeof(idt80),idtr.base+0x80*sizeof(idt80))
```

This function returns a pointer to the Interrupt Descriptor of the System Call Function (`int $0x80`). The program is now able to compute the entry point of the System Call function within kernel memory. This is accomplished by the following code: `old80 = idt80.off1 | (idt80.off2 << 16);`. However, this entry point does not provide the actual memory location that needs to be overwritten by the SuckIT rootkit to redirect any system calls to a malicious system call table that is created by the rootkit. We can examine the System Call Function assembly code within the kernel image (`vmlinux`) loaded at boot up by utilizing the resident code debugger (`gdb`: the GNU debugger) that exists within Red Hat Linux [78].

A specific system call function is invoked by the following:

```
call *sys_call_table(%eax,4).
```

The %eax register contains the number of the specific system call that is being called by the user program. Each entry in the system call table is four bytes long. To find the address of the system call that is to be invoked it is necessary to multiply the system call number (value stored in %eax register) by 4 (address size for 32 bit address) and add the result to the initial address of the system call table [83]. By examining this dump code, we see that the assembly code at

```
$ gdb -q /boot/vmlinux
```

```
(gdb) disass system_call
Dump of assembler code for function system_call:
0xc01070fc <system_call>:      push    %eax
0xc01070fd <system_call+1>:     cld
0xc01070fe <system_call+2>:     push    %es
0xc01070ff <system_call+3>:     push    %ds
0xc0107100 <system_call+4>:     push    %eax
0xc0107101 <system_call+5>:     push    %ebp
0xc0107102 <system_call+6>:     push    %edi
0xc0107103 <system_call+7>:     push    %esi
0xc0107104 <system_call+8>:     push    %edx
0xc0107105 <system_call+9>:     push    %ecx
0xc0107106 <system_call+10>:    push    %ebx
0xc0107107 <system_call+11>:    mov     $0x18,%edx
0xc010710c <system_call+16>:    mov     %edx,%ds
0xc010710e <system_call+18>:    mov     %edx,%es
0xc0107110 <system_call+20>:    mov     $0xffffe000,%ebx
0xc0107115 <system_call+25>:    and     %esp,%ebx
0xc0107117 <system_call+27>:    testb  $0x2,0x18(%ebx)
0xc010711b <system_call+31>:    jne     0xc010717c <tracesys>
0xc010711d <system_call+33>:    cmp     $0x100,%eax
0xc0107122 <system_call+38>:    jae     0xc01071a9 <badsys>
0xc0107128 <system_call+44>:    call    *0xc02d1890(,%eax,4)
0xc010712f <system_call+51>:    mov     %eax,0x18(%esp,1)
0xc0107133 <system_call+55>:    nop
End of assembler dump.
```

```
(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc02d1890
(gdb) x/xw (system_call+44)
0xc0107128 <system_call+44>: 0x908514ff
```

location 0xc0107128 (<system\_call+44>: call \*0xc02d1890(,%eax,4)) corresponds to

this command since we have also demonstrated that the value stored at the system call table = 0xc02d1890. We now wish to examine the memory at location <system\_call+44>. We utilize the x/Format Address command within gdb to do this. The exact format used is: (gdb) x/xw (system\_call+44) where xw – hex format word size [78]. The output of this command is 0x908514ff which is opcode in little endian format. The opcode 0xff 0x14 0x85 0x<address of the System Call Table> matches to the pattern ‘call \*some address (,%eax, 4)’ . This opcode pattern gives the SuckIt toolkit a specific pattern to search for within /dev/kmem. The address that follows this series of opcode is then changed by SuckIT to the address of the new System Call Table that the toolkit creates. Current LKM detectors do not check the consistency of the int \$0x80 function [78]. We find this to be significant because we propose that like SuckIT, one can query the int \$0x80 function to retrieve the current pointer to the System Call Table that is in use within the kernel and then check the integrity of this System Call Table to determine if this system has been infected with a kernel level toolkit of either type.

We have analyzed of the opcode series /xff/x14/x85/ to be sure that this will consistently be the opcode that SuckIT will need to search for to find the correct spot to modify the pointer to the System Call Table within /dev/kmem. According to the description of the Instruction Set of the INTEL Embedded Pentium ® Processor Family, the Opcode for the Call Instruction that we have seen from the disassembly of the system\_call function is as follows:

<u>Opcode</u>	<u>Instruction</u>	<u>Description</u>
FF/2	CALL r/m32	Call near, absolute indirect, address given in r/m32

The first opcode: xff, symbolizes the CALL instruction. The second opcode: x14, is in

the ModR/M byte of the instruction and symbolizes that a SIB byte follows this byte. The third opcode; x85, is in the SIB byte and symbolizes the 32 addressing format that is to be used, in this case [EAX\*4]. This series of opcode should not change between kernel versions as long as the INTEL Embedded Pentium ® Processor is used in the hardware platform[81].

A problem with using gdb to view this data is that the *vmlinux* kernel image that is used as input may not be an actual representation of what is currently loaded in the kernel. A kernel level rootkit may modify the kernel without changing any of the system files that are resident on the computer's file system to include the *vmlinux* file. A check of the current ktext segment against a previously archived clean version of the ktext will indicate that the kernel may have been compromised in some fashion. You will still be able to determine that the system call table has been tampered with by comparing the address of the system call table that is returned from querying the Interrupt Descriptor Table using the *sidt* assembly language command and comparing this value against the value that is retrieved from the *vmlinux* file and/or the address of the System Call Table (*sys\_call\_table*) that is stored in */boot/System.map* if these files are available. It is possible to view the actual data that is loaded into the kernel by using a program such as *kdb*, which is a kernel level debugger. If this program is available it is very easy to examine the kernel memory to view modifications. The following is an example of using *kdb* to display the instructions stored at a location in kernel memory:

```
kdb> id 0xc0107128
0xc0107128      system_call+0x2c: call *0xc02d1890( ,%eax,4)
```

The following example of *kdb* displays the contents of kernel memory stored at a particular location:

```
kdb> md 0xc0107128
0xc0107128      908514ff  89c02d18  90182444  147b83f0
```

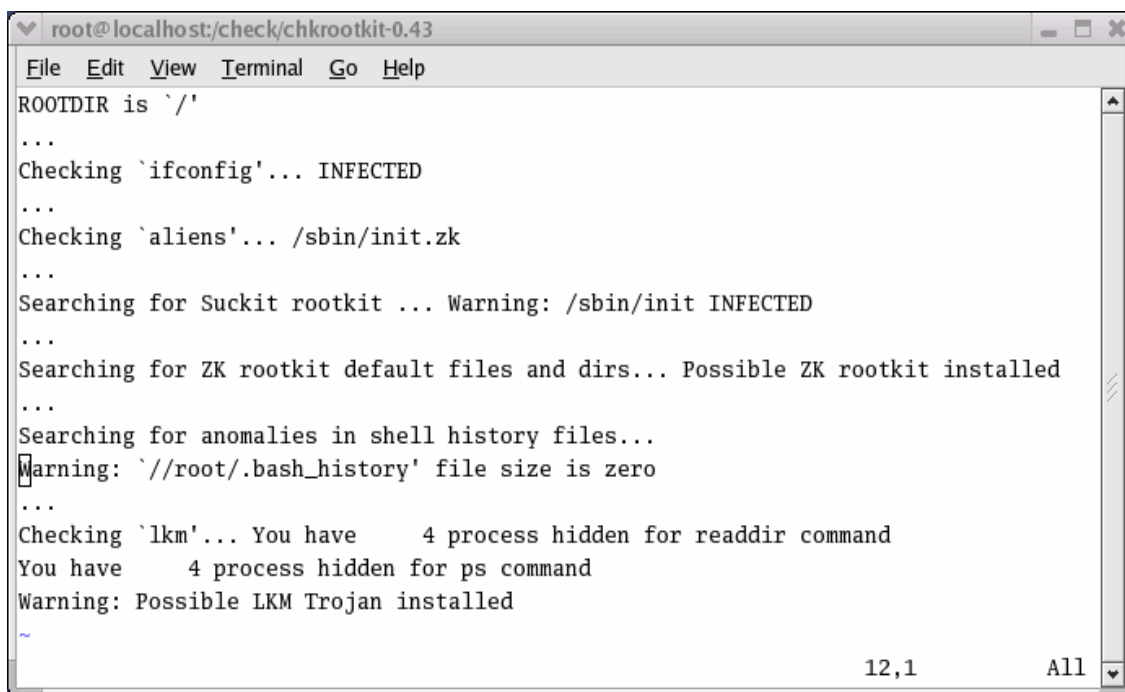
## A.9 The zk rootkit

The zk rootkit was developed by zaRwT@zaRwt.net. The documentation for this rootkit states that many of the features concerning patching of the kernel (/dev/kmem “Patching”) were borrowed from SuckIT. Therefore, we would expect that it is possible to detect the zk rootkit using the methods that we have already presented. However, the documentation talks about additional features that are different from what is contained in SuckIT. Our preliminary belief is that zk is a modification to the already existing SuckIT rootkit.

We once again followed our methodology in conducting an analysis of this rootkit. The standard initial steps were taken prior to infecting a system with this rootkit to include running the AIDE file integrity check program, copying the kernel text code segment, and running the chkrootkit program.

The AIDE program does detect that a file has been modified by this rootkit. This is different from the behavior of the AIDE program on a system that has been infected with SuckIT, which did not detect any file modifications. The file detected as being modified is the /sbin/ifconfig file. The AIDE program also detects that the file /sbin/inetcfg has been added to the system. The /sbin/init file is also detected as being modified by the most recent version (v-0.43) of the chkrootkit program as was the case with the SuckIT rootkit. In addition, this most recent version of chkrootkit detects that the system is infected with the zk rootkit. It accomplishes this by searching for a particular file (load.zk) that is appended with the default file hiding string (‘.zk’). Any files that contain this string is hidden from general display by the kernel level rootkit. These files

are still accessible if you access them via their exact filename. If the file hiding string is changed to something other than '.zk' then chkrootkit will not detect them. The chkrootkit program also detects other anomalies with the infected system. These are potential deltas ( $\nabla$ ) that can be used to characterize the zk rootkit. Figure 63 shows the results of running the chkrootkit program (v-0.43) on the system that has been infected with the zk rootkit.



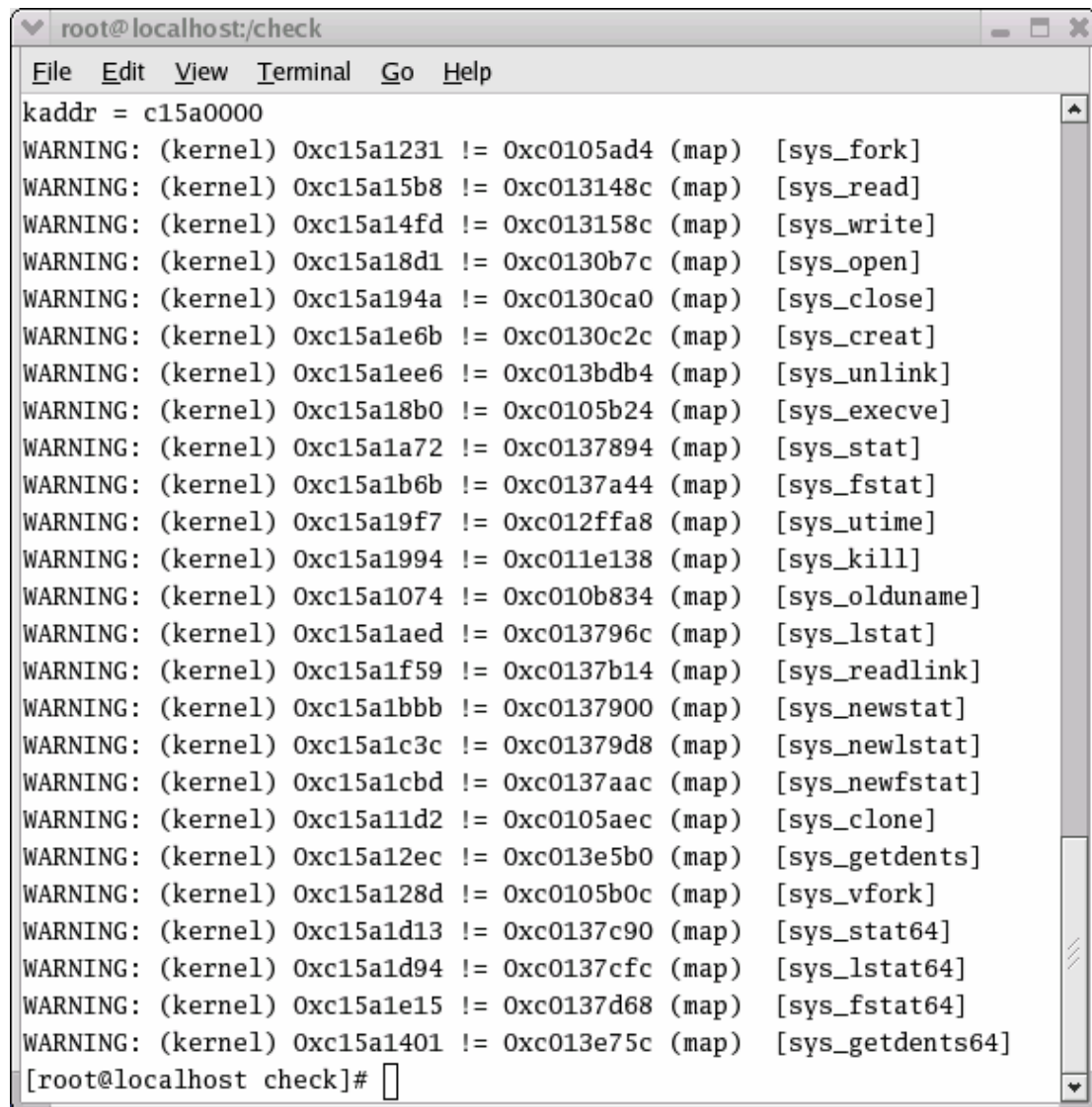
```
root@localhost:/check/chkrootkit-0.43
File Edit View Terminal Go Help
ROOTDIR is '/'
...
Checking `ifconfig'... INFECTED
...
Checking `aliens'... /sbin/init.zk
...
Searching for Suckit rootkit ... Warning: /sbin/init INFECTED
...
Searching for ZK rootkit default files and dirs... Possible ZK rootkit installed
...
Searching for anomalies in shell history files...
Warning: `//root/.bash_history' file size is zero
...
Checking `lkm'... You have      4 process hidden for readdir command
You have      4 process hidden for ps command
Warning: Possible LKM Trojan installed

12,1 All
```

**Figure 63: chkrootkit results on zk infected system**

Like the SuckIT rootkit, the zk rootkit changes 25 system calls. These 25 system calls are the same as those that are changed by SuckIT. These new replacement system calls are written into kernel memory in the same order as those from SuckIT, however, the zk replacement system calls, with the exception of `execve`, differ in size from the SuckIT replacement system calls. An examination of the zk source code indicates that the

developer changed some of the code in the portions of the zk rootkit originally based on the SuckIT rootkit. This is the most likely reason for the change in system call sizes and is another potential delta ( $\nabla$ ). Figure 64 shows the results of checking the system call table that is used by a target system infected with the zk rootkit.



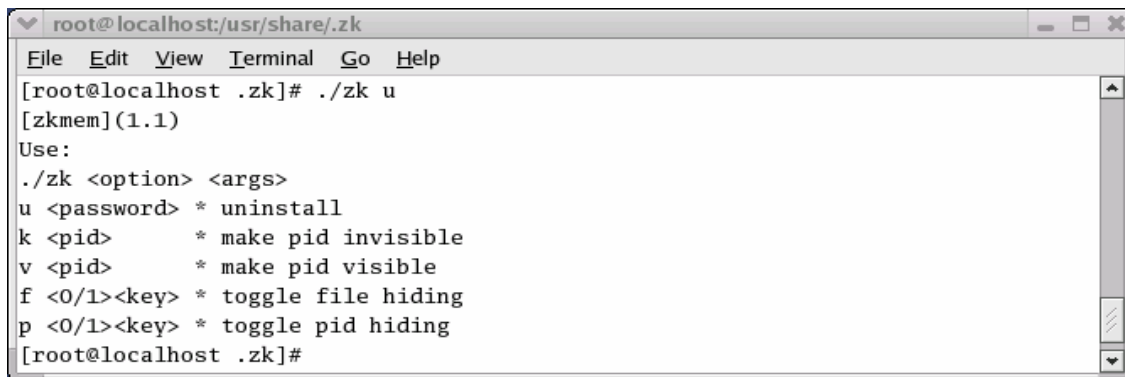
```
root@localhost:/check
File Edit View Terminal Go Help
kaddr = c15a0000
WARNING: (kernel) 0xc15a1231 != 0xc0105ad4 (map) [sys_fork]
WARNING: (kernel) 0xc15a15b8 != 0xc013148c (map) [sys_read]
WARNING: (kernel) 0xc15a14fd != 0xc013158c (map) [sys_write]
WARNING: (kernel) 0xc15a18d1 != 0xc0130b7c (map) [sys_open]
WARNING: (kernel) 0xc15a194a != 0xc0130ca0 (map) [sys_close]
WARNING: (kernel) 0xc15a1e6b != 0xc0130c2c (map) [sys_creat]
WARNING: (kernel) 0xc15a1ee6 != 0xc013bdb4 (map) [sys_unlink]
WARNING: (kernel) 0xc15a18b0 != 0xc0105b24 (map) [sys_execve]
WARNING: (kernel) 0xc15a1a72 != 0xc0137894 (map) [sys_stat]
WARNING: (kernel) 0xc15a1b6b != 0xc0137a44 (map) [sys_fstat]
WARNING: (kernel) 0xc15a19f7 != 0xc012ffa8 (map) [sys_utime]
WARNING: (kernel) 0xc15a1994 != 0xc011e138 (map) [sys_kill]
WARNING: (kernel) 0xc15a1074 != 0xc010b834 (map) [sys_olduname]
WARNING: (kernel) 0xc15a1aed != 0xc013796c (map) [sys_lstat]
WARNING: (kernel) 0xc15a1f59 != 0xc0137b14 (map) [sys_readlink]
WARNING: (kernel) 0xc15a1bbb != 0xc0137900 (map) [sys_newstat]
WARNING: (kernel) 0xc15a1c3c != 0xc01379d8 (map) [sys_newlstat]
WARNING: (kernel) 0xc15a1cbd != 0xc0137aac (map) [sys_newfstat]
WARNING: (kernel) 0xc15a11d2 != 0xc0105aec (map) [sys_clone]
WARNING: (kernel) 0xc15a12ec != 0xc013e5b0 (map) [sys_getdents]
WARNING: (kernel) 0xc15a128d != 0xc0105b0c (map) [sys_vfork]
WARNING: (kernel) 0xc15a1d13 != 0xc0137c90 (map) [sys_stat64]
WARNING: (kernel) 0xc15a1d94 != 0xc0137cfc (map) [sys_lstat64]
WARNING: (kernel) 0xc15a1e15 != 0xc0137d68 (map) [sys_fstat64]
WARNING: (kernel) 0xc15a1401 != 0xc013e75c (map) [sys_getdents64]
[root@localhost check]#
```

Figure 64: kern\_check results on system infected with the zk rootkit

The ktext files differ on a target system that is infected with zk which is the same behavior as SuckIT. The zk rootkit performs in a manner similar to SuckIT upon system

reboot in that each subsequent ktext file will differ from the previous version.

Unlike the SuckIT rootkit, we were not able to uninstall the zk rootkit program. This is another indication that SuckIT and zk are not the same. We then looked to identify this specific  $\nabla$  between these two programs concerning the uninstall process. One of the first things that we noticed is that when we try to run the uninstall command on the zk rootkit (`# ./zk u`), a usage statement is output to the screen and the program does not uninstall as indicated in figure 65. This is not the case with SuckIT, the uninstall program for SuckIT (`# ./sk u`) is successful.



```
root@localhost:/usr/share/.zk
File Edit View Terminal Go Help
[root@localhost .zk]# ./zk u
[zkmem](1.1)
Use:
./zk <option> <args>
u <password> * uninstall
k <pid>      * make pid invisible
v <pid>      * make pid visible
f <0/1><key> * toggle file hiding
p <0/1><key> * toggle pid hiding
[root@localhost .zk]#
```

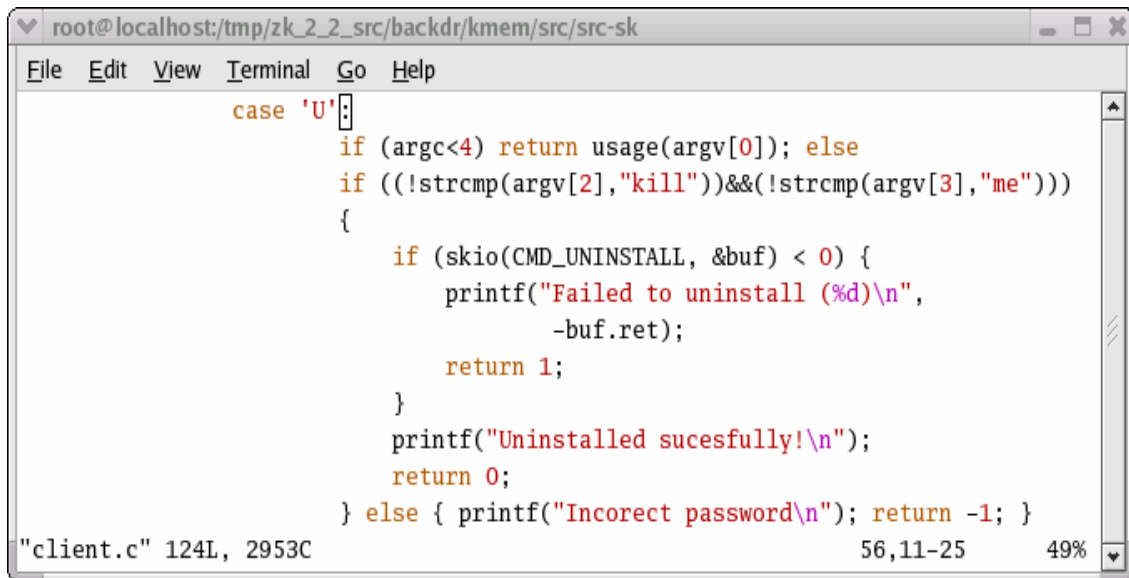
Figure 65: Unsuccessful uninstall of zk rootkit

To uninstall the zk rootkit, the usage statement indicates that a password must be used. There is no reference to this uninstall password within the zk rootkit documentation and there is no indication of how to set this password. We used the zk usage statement to try and identify a  $\nabla$ .

We conducted a *grep* search for the term ‘password’ within the source code directory for the zk rootkit. The results of this search indicate that the term ‘password’ exists within the `client.c` source code file. A file by the same name exists for the SuckIT rootkit. Comparing these two files using the resident *diff* command indicates that these



two files do in fact differ. We then conducted a more complete search on the zk client.c file. We identified a password 'kill me' within the client.c file. Figure 66 shows the results of this search.



```
root@localhost:tmp/zk_2_2_src/backdr/kmem/src/src-sk
File Edit View Terminal Go Help
    case 'U':
        if (argc<4) return usage(argv[0]); else
        if ((!strcmp(argv[2], "kill"))&&(!strcmp(argv[3], "me")))
        {
            if (skio(CMD_UNINSTALL, &buf) < 0) {
                printf("Failed to uninstall (%d)\n",
                    -buf.ret);
                return 1;
            }
            printf("Uninstalled sucesfully!\n");
            return 0;
        } else { printf("Incorect password\n"); return -1; }
"client.c" 124L, 2953C                                     56,11-25 49%
```

**Figure 66: Uninstall password for zk rootkit**

We were then able to successfully uninstall the zk rootkit by using the following command: # ./zk u kill me. Running the modified kern\_check program on the system indicates that the system is no longer infected.

The ability to examine both rootkits system allows you to continue to identify ∇'s, or differences between the two rootkits. The string 'kill me' can be used as a signature to detect instances of the zk rootkit. Other potential signatures can be identified from both rootkits in a similar manner.

## References

- [1] D. Dettrich, (2002, 5 JAN) “*Root Kits*” and *hiding files/directories/processes after a break-in*, [Online]. Available: <http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq>
- [2] E. Skoudis, *Counter Hack*, Upper Saddle River , NJ: Prentice Hall PTR: 2002, pp. 421-430.
- [3] <http://www.chkrootkit.org>, AUG 2002.
- [4] Thimbleby, S. Anderson, p. Cairns, “A Framework for Modeling Trojans and Computer Virus Infections,” *The Computer Journal*, vol. 41, no.7 pp. 444-458, 1998.
- [5] E. Cole, *Hackers Beware*, Indianapolis, In: New Riders, 2002, pp. 548-553.
- [6] O’Brian, D. Recognizing and Recovering from Rootkit Attacks. *Sys Admin* 5,11 (Nov 1996), pp. 8-20.
- [7] Silberschatz, P. Galvin, G. Gagne, *Applied Operating System Concepts*, New York, NY: John Wiley & Sons: 2003, p. 626.
- [8] S. Northcut, L. Zeltser, S. Winters, K. Kent Fredericks, R. Ritchey, *Inside Network Perimeter Security*. Indianapolis, In: New Riders, 2003, 283-286.
- [9] R. Lehti , “ The Aide Manual”, [www.cs.tut.fi/~rammer/aide/manual.html](http://www.cs.tut.fi/~rammer/aide/manual.html) , SEP 2002.
- [10] W. Venema, “Strangers in the Night – Finding the Purpose of an Unknown Program” *Dr. Dobb’s Journal*, November 2000, available at: <http://www.ddj.com/documents/s=879/ddj0011g/0011g.html>.

- [11] S. Forrest, S. Hofmeyer, A. Somayaji, "A Sense of Self for Unix Processes", in *Proc. 1996 IEEE Symposium on Security and Privacy*, , Los Angeles, 1996 pp. 120-128.
- [12] Samhain Labs, *The Basics— Subverting the Kernel*, <http://la-samha.de/library/rootkits /basics.html>, JAN 2003
- [13] Samhain Labs, *Detecting Kernel Rootkits*, <http://la-samha.de/library/rootkits/detect.html>, JAN 2003
- [14] Samhain Labs (email, 27 JAN 2003)
- [15] <http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk-4.1.tar.gz>, SEP 2002
- [16] <http://hysteria.sk/sd/f/suckit/>, JUN 2003
- [17] <http://www.oit..gatech.edu>, SEP 2002.
- [18] <http://security.gatech.edu>, SEP 2002.
- [19] C. Kuethe, "Through the Looking Glass: Finding Evidence of Your Cracker" *The Linux Gazette*, issue 36, Jan 1999, available at: <http://www.linuxgazette.com/issue36 /kuenthe.html>, AUG 2002.
- [20] R. Di Pietro, L. Mancini, "A Methodology of Computer Forensics Analysis" presented at the 2002 IEEE Workshop on Information Assurance, West Point, NY, 17-20 June 2002.
- [21] Levine, J., Owen, H., Culver, B., "A Methodology for Detecting New Binary Rootkit Exploits", presented at the 2003 IEEE SoutheastCon 2003, Ocho Rios, Jamaica, 4-6 April 2003.
- [22] Cohen, F., "Computer Viruses", *Computers & Security*. 6(1), pp. 22-35., 1987.
- [23] kad, "Handling the Interrupt Descriptor Table for fun and profit", *Phrack*, issue 59, article 4, July 28, 2002, available at <http://www.phrack.org>, SEP 2003.
- [24] Levine, J. (email, 24 March 2003)
- [25] T. Garfinkel, M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection", presented at Internet Society's 2003 Symposium on Network and Distributed System Security, San Diego, CA, 6-7 FEB 2003, OCT 2003.
- [26] Lawless, T. README, StMichael-LKM-0.11.tar.gz, 30 MAR 2002.

- [27] <http://www.antiserver.it/Backdoor-Rootkit/>, (adore-0.38.tar.gz), OCT 2003
- [28] Samhain Labs, <http://la-samha.de/samhain/index.html>, July 2003
- [29] Chuvakin, A. “Ups and Downs of UNIX/Linux Host-Based Security Solutions”, *login: The Magazine of USENIX & SAGE*, vol.28, number 2, April 2003
- [30] Rutkowski, J., “Execution Path Analysis: finding kernel based rootkits”, *Phrack*, issue 59, article 10, July 28, 2002, available at <http://www.phrack.org>, OCT 2003.
- [31] Rutkowski, J., “Advanced Windows 2000 Rootkit Detection (Execution Path Analysis)”, present at Black Hat USA 2003, Las Vegas, NV 28-31 July 2003, OCT 2003.
- [32] <http://www.derkeiler.com/Newsgroups/comp.os.linux.security/2002-02/0794.html>, AUG 2003.
- [33] Karresand, M., “A Proposed Taxonomy of Software Weapons”, Master’s Thesis, Linkoping University, Linkoping, Sweden, Dec 2002, SEP 2003.
- [34] Zovi, D., “Kernel Rootkits”, <http://www.cs.unm.edu/~ghandi/lkr.pdf>, 3 July 2001, OCT 2003.
- [35] <http://oss.sgi.com/projects/kdb>, OCT 2003
- [36] <http://kgdb.sourceforge.net/>. OCT 2003
- [37] [http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/Security/chapter\\_3\\_section\\_7.html](http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/Security/chapter_3_section_7.html), OCT 2003.
- [38] [http://publib16.boulder.ibm.com/pseries/en\\_US/files/aixfiles/mem.htm](http://publib16.boulder.ibm.com/pseries/en_US/files/aixfiles/mem.htm), OCT 2003.
- [39] <http://www.securityfocus.com/archive/1/273002>, OCT 2003.
- [40] D. Bovet, M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA: O’Reilly & Associates, 2003, pp. 62-63.
- [41] D. Bovet, M. Cesati, p 303.
- [42] Pragmatic/THC, “(nearly) Complete Linux Loadable Kernel Modules”, March 1999, available at [http://packetstormsecurity.org/docs/hack/LKM\\_HACKING.html](http://packetstormsecurity.org/docs/hack/LKM_HACKING.html), JULY 2003.

- [43] <http://sourceforge.net/projects/biew/>, OCT 2003.
- [44] S. Northcutt, J. Novak, *Network Intrusion Detection An Analyst's Handbook* . Indianapolis, New Riders, 2001, p. 177.
- [45] <http://www.intel.com/design/intarch/techinfo/pentium/opcode.htm>, OCT 2002
- [46] Levine, J., LaBella, R., Owen, H., Contis, D., Culver, B., "The Use of a Honeynet to Detect Exploits Across Large Enterprise Networks", presented at the 2003 IEEE Workshop on Information Assurance, West Point, NY, 18-20 June 2003.
- [47] The Honeynet Project, *Know Your Enemy*, Indianapolis, IN: Addison-Wesley, 2002, pp. 12-17.
- [48] The Honeynet Project, *Know Your Enemy*, p. 20.
- [49] L. Spitzner, *Honeypots- Tracking Hackers*, Indianapolis, IN: Addison-Wesley, 2003, pp. 242-261.
- [50] <http://project.honeynet.org/papers /honeynet/tools/rc.firewall>, JULY 2002.
- [51] <http://www.snort.org>. JULY 2002.
- [52] <http://www.xcdroast.de>, JULY 2002.
- [53] <http://enterprisesecurity.symantec.com/content.cfm?articleid=1539>, NOV 2003.
- [54] <http://www.cert.org/advisories/CA-2003-20.html>, NOV 03.
- [55] L. Spitzner, p. 69.
- [56] [http://www.linuxsecurity.com/feature\\_stories/feature\\_story-141.html](http://www.linuxsecurity.com/feature_stories/feature_story-141.html), NOV 2003.
- [57] <http://www.packetfu.org/hpa.html>, NOV 2003.
- [58] <http://www.honeylux.org.lu/project/honeyluxR1/result/sub01/report/hax.html>, AUG 2003.
- [59] <http://packetstormsecurity.nl/trojans/indexdate.shtml/clean-osf.8759.tgz /README>, DEC 2003.
- [60] S. Hawkins, "Understanding the Attackers Toolkit" *The SANS Institute Reading Room*, January 13, 2001, <http://rr.sans.org/linux/toolkit.php>. AUG 2002.
- [61] <http://packetstormsecurity.org/UNIX/penetration/rootkits>, AUG 2002.

- [62] <http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk-4.1.tar.gz>, SEP 2002.
- [63] util-linux-2.10f-src.rpm.at <http://rpmfind.net>, SEP 2002.
- [64] <ftp://i17linuxb.ists.pwr.wroc.pl/pub/linux/shadow/shadow.lsm>, SEP 2002
- [65] <http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk5.src.tar.gz> DEC 2003.
- [66] E. Skoudis, *Counter Hack*, p 430.
- [67] T. Miller, “Analysis of the T0rn rootkit”, <http://www.sans.org/y2k/t0rn.htm>, SEP 2002.
- [68] <http://www.securityfocus.com/news/813>, DEC 2003.
- [69] <http://www.sans.org/y2k/lion.htm>, DEC 2003.
- [70] R. Zuver, “A Thousand Heads are Better Than One – The Present and Future of Distributed Intrusion Detection”, <http://www.sans.org/rr/papers/30/361.pdf>, DEC 2003.
- [71] T. Miller, “Analysis of the KNARK rootkit”, <http://online.securityfocus.com/guest/4871>, Nov 2002.
- [72] <http://packetstorm.linuxsecurity.org>, NOV 2002.
- [73] [http://linux.fedwaycc.org/knark/creed\\_interview1.html](http://linux.fedwaycc.org/knark/creed_interview1.html), NOV 2002.
- [74] <http://www.sans.org/resources/idfaq/knark.php>, MAR 2003.
- [75] <http://bvi.sourceforge.net/>, NOV 2002.
- [76] <http://www.intel.com/design/pentium4/manuals/245471.htm>, NOV 2003.
- [77] <http://www.s0ftpj.org/docs/lkm.htm>, DEC 2003.
- [78] s.d., devik, *Linux-on-the-fly kernel patching without LKM*, <http://www.pharck.org/phrack/58/p58-0x07>, 12 Dec 2002.
- [79] D. Bovet, M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA: O’Reilly & Associates, 2003, pp. 304-306.
- [80] <http://mail.nl.linux.org/kernelnewbies/2002-06/msg00204.html>, DEC 2003.

- [81] <http://www.intel.com/design/intarch/techinfo/pentium/instrefs.htm#96030>, Jul 2003.
- [82] D. Bovet, M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2003, pp. 304-306.
- [83] <http://www.honeynet.org>, JUN 2002.

## Vita

John Levine was born in the United States of America in the 1960's. He received a B.S. from the United States Military Academy, West Point New York in the 1980's. He earned a M.S. from the Naval Postgraduate School, Monterey, California and a M.M.A.S. from the US Army Command and General Staff College, Ft Leavenworth Kansas in the 1990's. He joined the doctoral program of the School of Electrical and Computer Engineering at the Georgia Institute of Technology in 2001. In May of 2004 he completed his doctoral studies and graduated from the Georgia Institute of Technology with his Ph.D. in Electrical and Computer Engineering. His research interests include computer security and Honeynets.